INTER LISP

INTERLISP REFERENCE MANUAL

BY **WARREN TEITELMAN**

contributions by:

A. K. HARTLEY
J. W. GOODWIN
BOLT BERANEK & NEWMAN

D. G. BOBROW
P. C. JACKSON
L. M. MASINTER
XEROX PALO ALTO RESEARCH CENTER



Acknowledgements and Background

INTERLISP (formerly BBN LISP) has evolved from a succession of LISP systems that began with a LISP designed and implemented for the DEC PDP-1 by D. G. Bobrow and D. L. Murphy[1] at Bolt, Beranek and Newman in 1966, and documented by D. G. Bobrow. An upwards compatible version of this LISP was implemented for the SDS 940 in 1967, by Bobrow and Murphy. This system contained the seeds for many of the capabilities and features of the current system: a compatible compiler and interpreter,[2] uniform error handling, an on-line LISP oriented editor,[3] sophisticated debugging facilities,[4] etc. 940 LISP was also the first LISP system to demonstrate the feasibility of using software paging techniques and a large virtual memory in conjunction with a list-processing system [Bob2]. DWIM, the Do-What-I-Mean error correction facility, was introduced into the system in 1968 by W. Teitelman [Tei2], who was also responsible for documentation for the 940 LISP system.

---

1 D. G. Bobrow is currently at Xerox Palo Alto Research Center (PARC), D. L. Murphy is with Digital Equipment Corp.

2 The preliminary version of the compiler was written by L. P. Deutsch, now at Xerox PARC. This was considerably modified and extended by D. L. Murphy before producing the final working version.

3 The original idea of a LISP oriented structure editor belongs to L. P. Deutsch. The editor in its current form was written by W. Teitelman, now of Xerox PARC.

4 Designed and implemented by W. Teitelman.

In 1970, an upwards compatible version of 940 LISP called BBN LISP[6] was designed for the PDP-10 by D. G. Bobrow, D. L. Murphy, A. K. Hartley, and W. Teitelman, and implemented by Hartley with assistance from Murphy. A. K. Hartley was also responsible for modifying the 940 LISP compiler to generate code for the PDP-10. BBN-LISP ran under TENEX, a sophisticated time sharing system for the PDP-10 designed and implemented by D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, T. R. Strollo, and R. S. Tomlinson.[Bob1] With hardware paging and 256K of virtual memory provided by TENEX, it became practical to provide extensive and sophisticated interactive user support facilities, such as the programmer's assistant [Tei4], CLISP [Tei5], and a more sophisticated DWIM, all of which were designed and developed by W. Teitelman. In 1971, the block compiler was designed and implemented by D. G. Bobrow. The BBN-LISP Manual [Tei3] was written by W. Teitelman, with contributions from A. K. Hartley and from J. W. Goodwin, who also wrote TRANSOR and the special arithmetic functions, as well as a number of other utility functions. The name of the system was changed from BBN-LISP to INTERLISP in 1973, when the maintenance and development of the system evolved into a joint effort between Bolt Beranek and Newman, and Xerox Palo Alto Research Center. The INTERLISP reference manual was written by W. Teitelman, with contributions from A. K. Hartley, J. W. Goodwin, and D. G. Bobrow. The cover was designed by Alice R. Fikes.

INTERLISP is currently the LISP system used at Bolt Beranek and Newman, Xerox Palo Alto Research Center, Stanford Research Institute Artificial Intelligence Center, Information Sciences Institute, and the Dendral Project at Stanford University, in addition to being available at Computer Corporation of America

---

6 The design, construction and documentation for BBN LISP was sponsored by the Information Processing Techniques Section of the Advanced Research Project Agency, as was all of the subsequent work on the system that was performed at BBN. Since March 1972, the contributions made to the development of the system by W. Teitelman, including the preparation of this manual, were sponsored by Xerox Palo Alto Research Center.

and Case Institute of Technology. The total user community now comprises approximately one hundred users.

INTERLISP is a continuously evolving system, both in response to complaints, suggestions, and requests of the many users scattered throughout the ARPA network, as well as the long range goals of the individuals primarily responsible for the system, which are currently:

Person	Responsible for
W. Teitelman Xerox Palo Alto Research Center 3180 Porter Dr. Palo Alto, Calif. 94304	User Facilities: i.e., prettyprint, editor, break and trace, advising, printstructure, DWIM, CLISP, programmer's assistant.
A. K. Hartley Bolt Beranek & Newman 50 Moulton St. Cambridge, Mass. 02138	Basic System: i.e., interpreter, input-output, garbage collector; plus all SUBRS, i.e. hand-coded machine language functions such as PRINT, CONS, PROG, GO, etc.; plus compiler.
J. W. Goodwin Bolt Beranek & Newman 50 Moulton St. Cambridge, Mass. 02138	Special Arithmetic Functions: e.g. LOG, SIN, SQRT, etc.; plus functions for accessing TENEX capabilities such as SUBSYS, FILDIR, et al.; plus TRANSOR as well as various utility functions such as LOADFNS, SORT, etc. (as indicated in the text of this manual).

* * *

The preparation of this manual has involved the efforts of several persons at Xerox PARC, whom I specifically want to mention, and to express my appreciation for their support through this arduous, and at times seemingly endless task. Thank you Suzan (Jerome), Janet (Farness), Peter (Deutsch), Bob (Walker), and Larry (Tesler). I couldn't have done it without you.

Warren Teitelman
Palo Alto
December, 1973.

TABLE OF CONTENTS

TABLE OF CONTENTS (cont.)

page

SECTION 8: Function Definition and Evaluation

SECTION 9: The INTERLISP Editor

SECTION 10: Atom, String, Array, and Storage Manipulation

SECTION 11: Functions with Functional Arguments

SECTION 12: Variable Bindings and Pushdown List Functions

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

page

SECTION 17: Automatic Error Correction - The DWIM Facility

SECTION 18: The Compiler and Assembler

SECTION 19: Advising

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

SECTION 1

INTRODUCTION

This document is a reference manual for INTERLISP, a LISP system currently implemented on the DEC PDP-10 under the BBN TENEX time sharing system.[Bob1] INTERLISP[1] is designed to provide the user access to the large virtual memory allowed by TENEX, with a relatively small penalty in speed (using special paging techniques described in [Bob2]). Additional data types have been added, including strings, arrays, and hash association tables (hash links) (Sections 7 and 10). The system includes a compatible compiler (Section 18) and interpreter. Machine code can be intermixed with INTERLISP expressions via the assemble directive of the compiler. The compiler also contains a facility for "block compilation" which allows a group of functions to be compiled as a unit, suppressing internal names. Each successive level of computation, from interpreted through compiled, to block-compiled provides greater speed at a cost of debugging ease.

INTERLISP has been designed to be a good on-line interactive system. Some of the features provided include elaborate debugging facilities with tracing and conditional breakpoints (Section 15), and a sophisticated LISP oriented editor within the system (Section 9). Utilization of a uniform error processing through user accessible routines (Section 16) has allowed the implementation of DWIM, a Do-What-I-Mean facility, which automatically corrects many types of errors without losing the context of computation (Section 17). The CLISP

---

[1] INTERLISP (formerly BBN LISP) is the most recent incarnation in a succession of LISP systems. See Acknowledgements at front of manual.

facility (Section 23) extends the LISP syntax by enabling ALGOL-like infix operators such as +, -, *, /, =, ←, AND, OR, etc., as well as IF-THEN-ELSE statements and FOR-WHILE-DO statements. CLISP expressions are automatically converted to equivalent LISP forms when they are first encountered. CLISP also includes list construction operators, as well as a LISP oriented pattern match compiler.

A novel and useful facility of the INTERLISP system is the programmer's assistant (Section 22), which monitors and records all user inputs. The user can instruct the programmer's assistant to repeat a particular operation or sequence of operations, with possible modifications, or to UNDO the effects of specified operations. The goal of the programmer's assistant, DWIM, CLISP, etc. is to provide a programming environment which will "cooperate" with the user in the development of his programs, and free him to concentrate more fully on the conceptual difficulties and creative aspects of the problem he is trying to solve.

To aid in converting to INTERLISP programs written in other LISP dialects, e.g., LISP 1.5, Stanford LISP, we have implemented TRANSOR, a subsystem which accepts transformations (or can operate from previously defined transformations), and applies these transformations to source programs written in another LISP dialect, producing object programs which will run on INTERLISP (Appendix 1). In addition, TRANSOR alerts the programmer to problem areas that (may) need further attention. TRANSOR was used extensively in converting from 940 LISP to BBN-LISP on the PDP-10. A set of transformations is available for converting from Stanford LISP and LISP 1.5 to INTERLISP.

A complete format directed list processing system FLIP [Teil], is available for use within INTERLISP.

Although we have tried to be as clear and complete as possible, this document

is not designed to be an introduction to LISP. Therefore, some parts may only be clear to people who have had some experience with other LISP systems. A good introduction to LISP has been written by Clark Weissman [Wei1]. Although not completely accurate with respect to INTERLISP, the differences are small enough to be mastered by use of this manual and on-line interaction. Another useful introduction is given by Berkeley [Ber1] in the collection of Berkeley and Bobrow [Ber2].

Changes to this manual will be issued by replacing sections or pages, and reissuing the index and table of contents at periodic intervals. In addition, the manual will be maintained on-line, and up to date versions of any or all chapters will be available in the form of TENEX files from W. Teitelman at Xerox PARC.

Bibliography

[Ber1] Berkeley, E.C., "LISP, A Simple Introduction" in Berkeley, E.C. and Bobrow, D.G. [Ber2].

[Ber2] Berkeley, E.C., and Bobrow, D.G. (editors), The Programming Language LISP, its Operation and Applications, MIT Press, 1966.

[Bob1] Bobrow, D. G., Burchfiel, J. D., Murphy, D. L., and Tomlinson, R. S. "TENEX, a Paged Time Sharing System for the PDP-10", Communications of the ACM, March, 1972.

[Bob2] Bobrow, D.G., and Murphy, D.L. "The Structure of a LISP System Using Two Level Storage", Communications of the ACM, V15 3, March 1967.

[Bob3] Bobrow, D.G., and Wegbreit, B. "A Model and Stack Implementation for Multiple Environments" (to be published), Third International Joint Conference on Artificial Intelligence, August 1973.

[McC1] McCarthy, J. et al. LISP 1.5 Programmer's Manual, MIT Press, 1966.

[Mur1] Murphy, D.L. "Storage Organization and Management in TENEX", Proceedings of Fall Joint Computer Conference, December 1972.

[Smi1] Smith, D. "MLISP" Artificial Intelligence Memo No. 135 Stanford University, October 1970.

[Tei1] Teitelman, W. FLIP, A Format Directed List Processor in LISP, BBN Report, 1967.

[Tei2] Teitelman, W. "Toward a Programming Laboratory" in Walker, D. (ed.) International Joint Conference on Artificial Intelligence, May 1969.

[Tei3] Teitelman, W., Bobrow, D.G., Hartley, A.K. Murphy, D.L. BBN-LISP TENEX Reference Manual, Bolt Beranek and Newman, July 1971, first revision February 1972, second revision August 1972.

[Tei4] Teitelman, W. "Automated Programmering - The Programmer's Assistant", Proceedings of the Fall Joint Computer Conference, December 1972.

[Tei5] Teitelman, W. "CLISP - Conversational LISP", Third International Joint Conference on Artificial Intelligence, August 1973.

[Wei1] Weissman, C. LISP 1.5 Primer, Dickenson Press (1967).

SECTION 2

USING INTERLISP

2.1 Using the INTERLISP Manual - Format, Notation, and Conventions

The INTERLISP manual is divided into separate more or less independent sections. Each section is paginated independently, to facilitate issuing updates of sections. Each section contains an index to key words, functions, and variables contained in that section. In addition, there is a composite index for the entire manual, plus several appendices and a table of contents.

Throughout the manual, terminology and conventions will be offset from the text and typed in italics, frequently at the beginning of a section. For example, one such notational convention is:

The names of functions and variables are written in lower case and underlined when they appear in the text. Meta-LISP notation is used for describing forms.

Examples: member[x;y] is equivalent to (MEMBER X Y), member[car[x];FOO] is equivalent to (MEMBER (CAR X) (QUOTE FOO)). Note that in meta-LISP notation lower case variables are evaluated, upper case quoted.

. notation is used to distinguish between cons and list.

e.g., if x=(A B C), (FOO x) is (FOO (A B C)), whereas (FOO . x) is (FOO A B C). In other words, x is cadr of (FOO x) but cdr of (FOO . x). Similarly, y is caddr of (FOO x y), but cddr of (FOO x . y). Note that this

convention is in fact followed by the read program, i.e., (FOO . (A B C)) and (FOO A B C) read in as equal structures.

Other important conventions are:

TRUE in INTERLISP means not NIL.

The purpose of this is to allow a single function to be used both for the computation of some quantity, and as a test for a condition. For example, the value of member[x;y] is either NIL, or the tail of y beginning with x. Similarly, the value of or is the value of its first TRUE, i.e., non-NIL, expression, and the value of and is either NIL, or the value of its last expression.

Although most lists terminate in NIL, the occasional list that ends in an atom, e.g., (A B . C) or worse, a number or string, could cause bizarre effects. Accordingly, we have made the following implementation decision:

All functions that iterate through a list, e.g., member, length, mapc, etc. terminate by an nlistp check, rather than the conventional null-check, as a safety precaution against encountering data types which might cause infinite cdr loops, e.g., strings, numbers, arrays.

Thus, member[x;(A B . C)]=member[x;(A B)]

reverse[(A B . C)]=reverse[(A B)]

append[(A B . C);y]=append[(A B);y]

For users with an application requiring extreme efficiency,[1] we have provided fast versions of memb, last, nth, assoc, and length which compile open and

---

[1] A NIL check can be executed in only one instruction, an nlistp requires about 12, although both generate only one word of code.

terminate on NIL checks, and therefore may cause infinite cdr loops if given poorly formed arguments. However, to help detect these situations, fmemb, flast, fnth, fassoc, and flength all generate errors when interpreted if their argument ends in a non-list other than NIL, e.g. BAD ARGUMENT - FLAST.

Most functions that set system parameters, e.g., printlevel, linelength, radix, etc., return as their value the old setting. If given NIL as an argument, they return the current value without changing it.

All SUBRS, i.e., hand coded functions, such as read, print, eval, cons, etc., have 'argument names' (U V W) as described under arglist, Section 8. However, for tutorial purposes, more suggestive names are used in the descriptions of these functions in the text.

Most functions whose names end in p are predicates, e.g. numberp, tailp, exprp; most functions whose names end in q are nlambda's, i.e., do not require quoting their arguments, e.g., setq, defineq, nlsetq.

"x is equal to y" means equal[x;y] is true, as opposed to "x is eq to y" meaning eq[x;y] is true, i.e., x and y are the same identical LISP pointer.

When new literal atoms are created (by the read program, pack, or mkatom), they are provided with a function definition cell initialized to NIL (Section 8), a value cell initialized to the atom NOBIND (Section 16), and a property list initialized to NIL (Section 7). The function definition cell is accessed by the functions getd and putd described in Section 8. The value cell of an atom is car of the atom, and its property list is cdr of the atom. In particular, car of NIL and cdr of NIL are always NIL, and the system will resist attempts to change them.

The term list refers to any structure created by one or more conses, i.e. it does not have to end in NIL. For example, (A . B) is a list. The function listp, Section 5, is used to test for lists. Note that not being a list does not necessarily imply an atom, e.g., strings and arrays are not lists, nor are they atoms. See Section 10.

Many system functions have extra optional arguments for internal use that are not described in the writeups. For example, readline is described as a function of no arguments, but arglist[READLINE] returns (LINE LISPXFLG). In such cases, the user should just ignore the extra arguments.

INTERLISP departs from LISP 1.5 and other LISP dialects in that car *of a form is never evaluated*. In other words, if car of a form is not an atom with a function definition, and not a function object, i.e. a list car of which is LAMBDA, NLAMBDA, or FUNARG, an error is generated. apply or apply* (section 8) must be used if the name of a function is to be *computed* as for example, when functional arguments are applied.

2.2 Using the INTERLISP System on TENEX - An Overview

Call INTERLISP by typing LISP followed by a carriage return. INTERLISP will type an identifying message, the date, and a greeting, followed by a '←'. This prompt character indicates that the user is "talking to" the top level INTERLISP executive, called <u>evalqt</u>, (for historical reasons), just as '@' indicates the user is talking to TENEX. <u>evalqt</u> calls <u>lispx</u> which accepts inputs in either <u>eval</u> or <u>apply</u> format: if just one expression is typed on a line, it is evaluated; if two expressions are typed, the first is <u>apply</u>-ed to the second. <u>eval</u> and <u>apply</u> are described in section 8. In both cases, the value is typed, followed by ← indicating INTERLISP is ready for another input.

INTERLISP is normally exited via the function LOGOUT, i.e., the user types LOGOUT(). However, typing control-C at any point in the computation returns control immediately to TENEX. The user can then *continue* his program with no ill effects with the TENEX CONTINUE command, even if he interrupted it during a garbage collection. Or he can *reenter* his program at <u>evalqt</u> with the TENEX REENTER command. *The latter is DEFINITELY not advisable if the Control-C was typed during a garbage collection*. Typing control-D at any point during a computation will return control to <u>evalqt</u>. If typed during a garbage collection, the garbage collection will first be completed, and *then* control will be returned to INTERLISP's top level, otherwise, control returns immediately.

When typing to the INTERLISP <u>read</u> program, typing a control-Q will cause INTERLISP to print '##' and clear the input buffer, i.e., erase the entire line up to the last carriage return. Typing control-A erases the last character typed in, echoing a \ and the erased character. Control-A will not back up beyond the last carriage return. Control-O can be used to *immediately* clear

the output buffer, and rubout to *immediately* clear the input buffer.[2] In addition, typing control-U (in most cases) will cause the INTERLISP editor (Section 9) to be called on the expression being read, when the <u>read</u> is completed. Appendix 3 contains a list of all control characters, and a reference to that part of the manual where they are described.

Since the INTERLISP read program is normally line-buffered to make possible the action of control-Q,[3] the user must type a carriage return before any characters are delivered to the function requesting input, e.g.,

```
←E T↵          [4]
T
```

However, the read program *automatically* supplies (and prints) this carriage return when a matching right parenthesis is typed, making it unnecessary for the user to do so, e.g.,

```
←CONS(A B)
(A . B)
```

The INTERLISP read program treats square brackets as 'super-parentheses': a right square bracket automatically supplies enough right parentheses to match back to the last left square bracket (in the expression being read), or if none has appeared, to match the first left parentheses,

e.g., (A (B (C]=(A (B (C))),

(A [B (C (D] E)=(A (B (C (D))) E).

---

[2] The action of control-Q takes place when it is *read*. If the user has 'typed ahead' several inputs, control-Q will only affect at most the last line of input. Rubout however will clear the entire input buffer as soon as it is *typed*, i.e., even during a garbage collection.

[3] Except following control[T], see Section 14.

[4] '↵' is used throughout the manual to denote carriage-return.

% is the universal escape character for read. Thus to input an atom containing a syntactic delimiter, precede it by %, e.g. AB% (C or %%. See Section 14 for more details.

Most of the "basics" of on-line use of INTERLISP, e.g. defining functions, error handling, editing, saving your work, etc., are illustrated in the following brief console session. Underlined characters were typed by the user.

1. The user calls INTERLISP from TENEX, INTERLISP prints a date, and a greeting. The prompt character ← indicates the user is at the top level of INTERLISP.

2. The user defines a function, fact, for computing factorial of n. In INTERLISP, functions are defined via DEFINE or DEFINEQ, (Section 8). Functions may independently evaluate arguments, or not evaluate them, and spread their arguments, or not spread them (Section 4). The function fact shown here is an example of an everyday run-of-the-mill function of one argument, which is evaluated.

3. The user "looks" at the function definition. Function definitions in INTERLISP are stored in a special cell called the function definition cell, which is associated with the name of the function (Section 8). This cell is accessible via the two functions, getd and putd, (define and defineq use putd). Note that the user typed an input consisting of a single expression, i.e. (GETD (QUOTE FACT)), which was therefore interpreted as a form for eval. The user could also have typed GETD(FACT).

4. The user runs his function. Two errors occur and corrections are offered by DWIM (Section 17). In each case, the user indicates his approval, DWIM makes the correction, i.e. actually changes the definition of fact, and then continues the computation.

```
@LISP↵                                                              1

INTERLISP-10 11-17-73 ...

GOOD EVENING.
←DEFINEQ((FACT (LAMBDDA (N) (COND ((EQ N 0) NIL)                   2
(T (ITIMES N (FACTT (SUB1 N]
(FACT)
←(GETD (QUOTE FACT))                                                3
(LAMBDDA (N) (COND ((EQ N 0) NIL) (T (ITIMES N (FACTT (SUB1 N))))))
←FACT(3)                                                            4
LAMBDDA [IN FACT] -> LAMBDA ? YES↵
FACTT [IN FACT] -> FACT ? YES↵

NON-NUMERIC ARG                                                     5
NIL
IN ITIMES

(BROKEN)                                                            6
:BT↵
ITIMES
COND
FACT
COND
FACT
COND
FACT
**TOP**

:N↵                                                                 7
1
:EDITF(FACT)                                                        8
EDIT
*(R NIL 1)                                                          9
*OK↵                                                                10
FACT
:RETURN 1↵                                                          11
'BREAK' = 1
6
←PP FACT↵                                                           12

(FACT
  [LAMBDA (N)
    (COND
      ((EQ N 0)
       1)
      (T (ITIMES N (FACT (SUB1 N])
FACT                                                                13
←PRETTYDEF((FACT) FACT)                                             14
FACT.;1
```

5. An error occurs that DWIM cannot handle, and the system goes into a break. At this point, the user can type in expressions to be eval-ed or apply-ed exactly as at the top level. The prompt character ':' indicates that the user is in a break, i.e. that the context of his computation is available. In other words, the system is actually "within" or "below" the call to itimes in which the error occurred.

6. The user types in the break command, BT, which calls for a backtrace to be printed. In INTERLISP, interpreted and compiled code (see Section 18 for discussion of the compiler) are completely compatible, and in both cases, the name of the function that was called, as well as the names and values of its arguments are stored on the stack. The stack can be searched and/or modified in various ways (see Section 12).

   Break commands are discussed in Section 15, which also explains how the user can "break" a particular function, i.e. specify that the system go into a "break" whenever a certain function or functions are called. At that point the user can examine the state of the computation. This facility is very useful for debugging.

7. The user asks for the value of the variable n, i.e. the most recent value, or binding. The interpreter will search the stack for the most recent binding, and failing to find one, will obtain the top level value from the atom's value cell, which is car of the atom (Section 3). If there are no bindings, and the value cell contains the atom NOBIND, an unbound atom error is generated (Section 16).

8. The user realizes his error, and calls the editor to fix it. (Note that the system is *still* in the break.) The editor is described at length and in detail in Section 9. It is an extremely useful facility of INTERLISP. Section 9 begins with a simple introduction designed for the new user.

9. The user instructs the editor to replace all NIL's (in this case there is only one) by 1. The editor physically changes the expression it is operating on so when the user exits from the editor, his function, *as it is now being interpreted*, has been changed.

10. The user exits from the editor and returns to the break.

11. The user specifies the value to be used by itimes in place of NIL by using the break command RETURN. This causes the computation to continue, and 6 is ultimately returned as the value of the original input, fact(3).

12. The user prettyprints (Section 14) fact, i.e. asks it be printed with appropriate indentations to indicate structure. Prettyprint also provides a comment facility. Note that both the changes made to fact by the editor and those made by DWIM are in evidence.

13. The user writes his function on a file by using prettydef (Section 14), creating a TENEX file, FACT.;1, which when loaded into INTERLISP at a later date via the function load (Section 14), will cause fact to be defined as it currently is. There is also a facility in INTERLISP for saving and restoring an entire core image via the functions sysout and sysin (Section 14).

14. The user logs out, returning control to TENEX. However, he can still continue his session by re-entering INTERLISP via the TENEX REENTER or CONTINUE command.

Index for Section 2

Page Numbers

SECTION 3

DATA TYPES, STORAGE ALLOCATION, AND GARBAGE COLLECTION[1]

INTERLISP operates in an 18-bit address space. This address space is divided into 512 word pages with a limit of 512 pages, or 262,144 words, but only that portion of address space currently in use actually exists on any storage medium. INTERLISP itself and all data storage are contained within this address space. A pointer to a data element such as a number, atom, etc., is simply the address of the data element in this 18-bit address space.

3.1 Data Types

The data types of INTERLISP are lists, atoms, pnames, arrays, large and small integers, floating point numbers, string characters and string pointers. Compiled code and hash arrays are currently included with arrays.

In the descriptions of the various data types given below, for each data type, first the input syntax and output format are described, that is, what input sequence will cause the INTERLISP read program to construct an element of that type, and how the INTERLISP print program will print such an element. Next, those functions that construct elements of that data type are given. Note that some data types cannot be input, they can only be constructed, e.g. arrays. Finally, the format in which an element of that data type is stored in memory is described.

---

[1] This section was written by A. K. Hartley.

3.1.1 Literal Atoms

A literal atom is input as any string of non-delimiting characters that cannot be interpreted as a number. The syntatic characters that delimit atoms are space, end-of-line,[2] line-feed, % ( ) "] and [. However, these characters may be included in atoms by preceding them with the escape character %.

Literal atoms are printed by print and prin2 as a sequence of characters with %'s inserted before all delimiting characters (so that the atom will read back in properly). Literal atoms are printed by prin1 as a sequence of characters without these extra %'s. For example, the atom consisting of the five characters A, B, C, (, and D will be printed as ABC%(D by print and ABC(D by prin1. The extra %'s are an artifact of the print program; they are not stored in the atom's pname.

Literal atoms can be constructed by pack, mkatom, and gensym (which uses mkatom).

Literal atoms are unique. In other words, if two literal atoms have the same pname, i.e. print the same, they will *always* be the same identical atom, that is, they will always have the same address in memory, or equivalently, they will always be eq.[3] Thus if pack or mkatom is given a list of characters corresponding to a literal atom that already exists, they return a pointer to that atom, and do *not* make a new atom. Similarly, if the read program is given as input of a sequence of characters for which an atom already exists, it returns a pointer to that atom.

---

[2] An end-of-line character is transmitted by TENEX when it sees a carriage-return.

[3] Note that this is *not* true for strings, large integers, floating point numbers, and lists, i.e. they all can print the same without being eq.

A literal atom is a 3 word (36 bits) datum containing:

	Bits 0–17	Bits 18–35
WORD 1:	PROPERTY LIST (CDR)	TOP LEVEL BINDING (CAR)
WORD 2:	FUNCTION CALLING INSTRUCTION (0–35)	
WORD 3:	PNAME	RESERVED FOR FUNCTIONS ON FILES

FIGURE 3-1

Car of a literal atom, i.e. the right half of word 1, contains its top level binding, initially the atom NOBIND. Cdr of the atom is a pointer to its property list, initially NIL.

Word 2, the function definition cell, is a full 36 bit word, containing an instruction to be executed for calling the function associated with that atom, if any. The left half differs for different function types (i.e., EXPR, SUBR, or compiled code); the right half is a pointer to the function definition.[4]

The pname cell, the left half of the third word, contains a pointer to the pname of the atom. The remaining half word is reserved for an extension of INTERLISP to permit storing function definitions on files.

---

[4] This use of a full word saves some time in function calls from compiled code in that we do not need to look up the type of the function definition at call time.

3.1.2 Pnames

The pnames of atoms,[5] pointed to in the third word of the atom, comprise another data type with storage assigned as it is needed. This data type only occurs as a component of an atom or a string. It does not appear, for example, as an element of a list.

Pnames have no input syntax or output format as they cannot be directly referenced by user programs.

A pname is a sequence of 7 bit characters packed 5 to a word, beginning at a word boundary. The first character of a pname contains its length; thus the maximum length of a pname is 126 characters.

3.1.3 Numerical Atoms

Numerical atoms, or simply numbers, do not have property lists, value cells, functions definition cells, or explicit pnames. There are currently two types of numbers in INTERLISP: integers, and floating point numbers.

Integers

The input syntax for an integer is an optional sign (+ or -) followed by a

---

[5] All INTERLISP pointers have pnames, since we define a pname simply to be how that pointer is printed. However, only literal atoms and strings have their pnames explicitly stored. Thus, the use of the term pname in a discussion of data types or storage allocation means pnames of atoms or strings, and refers to a sequence of characters stored in a certain part of INTERLISP's memory.

sequence of digits, followed by an optional Q.[6] If the Q is present, the digits are interpreted in octal, otherwise in decimal, e.g. 77Q and 63 both correspond to the same integers, and in fact are indistinguishable internally since no record is kept of how integers were created.

The setting of radix (Section 14), determines how integers are printed: signed or unsigned, octal or decimal.

Integers are created by pack and mkatom when given a sequence of characters observing the above syntax, e.g. (PACK (LIST 1 2 (QUOTE Q))) = 10. Integers are also created as a result of arithmetic operations, as described in Section 13.

An integer is stored in one 36 bit word; thus its magnitude must be less than 2↑35.[7] To avoid having to store (and hence garbage collect) the values of small integers, a few pages of address space, overlapping the INTERLISP machine language code, are reserved for their representation. The small number pointer *itself*, minus a constant, is the value of the number. Currently the range of 'small' integers is -1536 thru +1535. The predicate smallp is used to test whether an integer is 'small'.

While small integers have a unique representation, large integers do not. In other words, two large integers may have the same value, but not the same address in memory, and therefore not be eq. For this reason the function eqp (or equal) should be used to test equality of large integers.

---

[6] and terminated by a delimiting character. Note that some data types are self-delimiting, e.g. lists.

[7] If the sequence of digits used to create the integer is too large, the high order portion is discarded. (The handling of overflow as a result of arithmetic operations is discussed in Section 13.)

Floating Point Numbers

A floating point number is input as a signed integer, followed by a decimal point, followed by another sequence of digits called the fraction, followed by an exponent (represented by E followed by a signed integer).[8] Both signs are optional, and either the fraction following the decimal point, or the integer preceding the decimal point may be omitted. One or the other of the decimal point or exponent may also be omitted, but at least one of them must be present to distinguish a floating point number from an integer. For example, the following will be recognized as floating point numbers:

5.   5.00   5.01   .3   5E2   5.1E2

5E-3   -5.2E+6

Floating point numbers are printed using the facilities provided by TENEX. INTERLISP calls the floating point number to string conversion routines[9] using the format control specified by the function fltfmt (Section 14). fltfmt is initialized to T, or free format. For example, the above floating point numbers would be printed free format as:

5.0   5.0   5.01   .3   500.0   510.0

.005   -5.2E6

Floating point numbers are also created by pack and mkatom, and as a result of arithmetic operations as described in section 13.

A floating point number is stored in one 36 bit word in standard PDP-10 format. The range is ±2.94E-39 thru ±1.69E38 (or 2↑-128 thru 2↑127).

---

8 and terminated by a delimiter.

9 Additional information concerning these conversions may be obtained from the TENEX JSYS Manual.

3.1.4 Lists

The input syntax for a list is a sequence (at least one)[10] of INTERLISP data elements, e.g. literal atoms numbers, other lists, etc. enclosed in parentheses or brackets. A bracket can be used to terminate several lists, e.g. (A (B (C], as described in Section 2.

If there are two or more elements in a list, the final element can be preceded by a . (delimited on both sides), indicating that cdr of the final node in the list is to be the element immediately following the ., e.g. (A . B) or (A B C . D), otherwise cdr of the last node in a list will be NIL.[11] Note that the input sequence (A B C . NIL) is thus equivalent to (A B C), and that (A B . (C D)) is thus equivalent to (A B C D). Note however that (A B . C D) will create a list containing the five literal atoms A B . C and D.

Lists are constructed by the primitive functions cons and list.

Lists are printed by printing a left parenthesis, and then printing the first element of the list,[12] then printing a space, then printing the second element, etc. until the final node is reached. Lists are considered to terminate when cdr of some node is not a list. If cdr of this terminal node is NIL (the usual case), car of the terminal node is printed followed by a right parenthesis. If cdr of the terminal node is *not* NIL, car of the terminal node is printed,

---

[10] () is read as the atom NIL.

[11] Note that in INTERLISP terminology, a list does *not* have to end in NIL, it is simply a structure composed of one or more conses.

[12] The individual elements of a list are printed using prin2 if the list is being printed by print or prin2, and by prin1 if the list is being printed by prin1.

followed by a space, a period, another space, cdr of the terminal node, and then the right parenthesis. Note that a list input as (A B C . NIL) will print as (A B C), and a list input as (A B . (C D)) will print as (A B C D). Note also that printlevel affects the printing of lists to teletype, and that carriage returns may be inserted where dictated by linelength, as described in Section 14.

A list is stored as a chain of list *nodes*. A list node is stored in one 36 bit word, the right half containing car of the list (a pointer to the first element of the list), and the left half containing cdr of the list (a pointer to the next node of the list).

3.1.5 Arrays

An array in INTERLISP is a one dimensional block of contiguous storage of arbitrary length. Arrays do not have input syntax; they can only be created by the function array. Arrays are printed by both print, prin2, and prin1, as # followed by the address of the array pointer (in octal). Array elements can be referenced by the functions elt and eltd, and set by the functions seta and setd, as described in Section 10.

Arrays are partitioned into four sections: a header, a section containing unboxed numbers, a section containing INTERLISP pointers, and a section containing relocation information. The last three sections can each be of arbitrary length (including 0); the header is two words long and contains the length of the other sections as indicated in the diagram below. The unboxed number region of an array is used to store 36 bit quantities that are not INTERLISP pointers, and therefore not to be chased from during garbage collections, e.g. machine instructions. The relocation informaion is used when the array contains the definition of a compiled function, and specifies which

locations in the *unboxed* region of the array must be changed if the array is moved during a garbage collection.

The format of an array is as follows:

HEADER	WORD 0	ADDRESS OF RELOCATION INFORMATION	LENGTH
	WORD 1	USED BY GARBAGE COLLECTOR	ADDRESS OF POINTERS
FIRST DATA WORD		NON-POINTERS	
		POINTERS	
		RELOCATION INFORMATION	

FIGURE 3-2

The header contains:

word 0 right - length of entire block=ARRAYSIZE+2.

left - address of relocation information relative to word 0 of block (> 0 if relocation information exists, negative if array is a hash array, 0 if ordinary array).

word 1 right - address of pointers relative to word 0 of block.

left - used by garbage collector.

3.1.6 Strings

The input syntax for a string is a ", followed by a sequence of any characters except " and %, terminated by a ". " and % may be included in a string by preceding them with the escape character %.

Strings are printed by print and prin2 with initial and final "'s, and %'s inserted where necessary for it to read back in properly. Strings are printed by prin1 without the delimiting "'s and extra %'s.

Strings are created by mkstring, substring, and concat.

Internally a string is stored in two parts; a string pointer and the sequence of characters. The INTERLISP pointer to a string is the address of the string pointer. The string pointer, in turn, contains the character position at which the string characters begin, and the number of characters. String pointers and string characters are two separate data types,[13] and several string pointers may reference the same characters. This method of storing strings permits the creation of a substring by creating a new string pointer, thus avoiding copying of the characters. For more details, see Section 10.

String characters are 7 bit bytes packed 5 to a word. The format of a string pointer is:

# OF CHARACTERS	5 * ADDRESS OF STRING + CHARACTER POSITION
0 14	15 35

FIGURE 3-3

The maximum length of a string is 32K (K=1024) characters.

---

[13] String characters are not directly accessible by user programs.

3.2 Storage Allocation and Garbage Collection

In the following discussion, we will speak of a quantity of memory being assigned to a particular data type, meaning that the space is reserved for storage of elements of that type. *Allocation* will refer to the process used to obtain from the already assigned storage a particular location for storing one data element.

A small amount of storage is assigned to each data type when INTERLISP is started; additional storage is assigned only during a garbage collection.

The page is the smallest unit of memory that may be assigned for use by a particular data type. For each page of memory there is a one word entry in a type table. The entry contains the data type residing on the page as well as other information about the page. The type of a pointer is determined by examining the appropriate entry in the type table.

Storage is allocated as is needed by the functions which create new data elements, such as cons, pack, mkstring. For example, when a large integer is created by iplus, the integer is stored in the next available location in the space assigned to integers. If there is no available location, a garbage collection is initiated, which may result in more storage being assigned.

The storage allocation and garbage collection methods differ for the various data types. The major distinction is between the types with elements of fixed length and the types with elements of arbitrary length. List *nodes*, atoms, large integers, floating point numbers, and string pointers are fixed length; all occupy 1 word except atoms which use 3 words. Arrays, pnames, and strings (string characters) are variable length.

Elements of fixed length types are stored so that they do not overlap page

boundaries. Thus the pages assigned to a fixed length type need not be adjacent. If more space is needed, any empty page will be used. The method of *allocating* storage for these types employs a free-list of available locations; that is, each available location contains a pointer to the next available location. A new element is stored at the first location on the free-list, and the free-list pointer is updated.[14]

Elements of variable length data types *are* allowed to overlap page boundaries. Consequently all pages assigned to a particular variable length type must be contiguous. Space for a new element is allocated following the last space used in the assigned block of contiguous storage.

When INTERLISP is first called, a few pages of memory are assigned to each data type. When the allocation routine for a type determines that no more space is available in the assigned storage for that type, a garbage collection is initiated. The garbage collector determines what data is currently in use and reclaims that which is no longer in use. A garbage collection may also be initiated by the user with the function reclaim (Section 10).

Data in use (also called active data) is any data that can be 'reached' from the currently running program (i.e., variable bindings and functions in execution) or from atoms. To find the active data the garbage collector 'chases' all pointers, beginning with the contents of the push-down lists and the components (i.e., car, cdr, and function definition cell) of all atoms with at least one non-trivial component.

---

[14] The allocation routine for list nodes is more complicated. Each page containing list nodes has a separate free list. First a page is chosen (see CONS for details), then the free list for that page is used. Lists are the only data type which operate this way.

When a previously unmarked datum is encountered, it is marked, and all pointers contained in it are chased. Most data types are marked using bit tables; that is tables containing one bit for each datum. Arrays, however, are marked using a half-word in the array header.

When the mark and chase process is completed, unmarked (and therefore unused) space is reclaimed. Elements of fixed length types that are no longer active are reclaimed by adding their locations to the free-list for that type. This free list allocation method permits reclaiming space without moving any data, thereby avoiding the time consuming process of updating all pointers to moved data. To reclaim unused space in a block of storage assigned to a variable length type, the active elements are compacted toward the beginning of the storage block, and then a scan of all active data that can contain pointers to the moved data is performed to update the pointers.

Whenever a garbage collection of any type is initiated,[15] unused space for all fixed length types is reclaimed since the additional cost is slight. However, space for a variable length type is reclaimed only when that type initiated the garbage collection.

If the amount of storage reclaimed for the type that initiated the garbage collection is less than the minimum free storage requirement for that type, the garbage collector will assign enough additional storage to satisfy the minimum free storage requirement. The minimum free storage requirement for each data may be set with the function minfs (Section 10). The garbage collector assigns additional storage to fixed length types by finding empty pages, and adding the appropriate size elements from each page to the free list. Assigning

---

[15] The 'type of a garbage collection' or the 'type that initiated a garbage collection' means either the type that ran out of space and called the garbage collector, or the argument to reclaim.

additional storage to a variable length type involves finding empty pages and moving data so that the empty pages are at the end of the block of storage assigned to that type.

In addition to increasing the storage assigned to the type initiating a garbage collection, the garbage collector will attempt to minimize garbage collections by assigning more storage to other fixed length types according to the following algorithm.[16] If the amount of active data of a type has increased since the last garbage collection by more than 1/4 of the minfs value for that type, storage is increased (if necessary), to attain the <u>minfs</u> value. If active data has increased by less than 1/4 of the <u>minfs</u> value, available storage is increased to 1/2 <u>minfs</u>. If there has been no increase, no more storage is added. For example, if the <u>minfs</u> setting is 2000 words, the number of active words has increased by 700, and after all unused words have been collected there are 1000 words available, 1024 additional words (two pages) will be assigned to bring the total to 2024 words available. If the number of active words had increased by only 300, and there were 500 words available, 512 additional words would be assigned.

3.3 Shared INTERLISP

The INTERLISP system initially obtained by the user is shared; that is, all active users of INTERLISP are actually using the same pages of memory. As a user adds to the system, private pages are added to his memory. Similarly, if the user changes anything in the original shared INTERLISP, for example, by advising a system function, a private copy of the changed page is created.

---

[16] We may experiment with different algorithms.

In addition to the swapping time saved by having several users accessing the same memory, the sharing mechanism permits a large saving in garbage collection time, since we do not have to garbage collect any data in the shared system, and thus do not need to chase from any pointers on shared pages during garbage collections.

This reduction in garbage collection time is possible because the shared system usually is not modified very much by the user. If the shared system is changed extensively, the savings in time will vanish, because once a page that was initially shared is made private, every pointer on it must be assumed active, because it may be pointed *to* by something in the shared system. Since every pointer on an initially shared but now private page can also point to *private* data, they must always be chased.

A user may create his own shared system with the function makesys. If several people are using the same system, making the system be shared will result in a savings in swapping time. Similarly, if a system is large and seldom modified, making it be shared will result in a reduction of garbage collection time, and may therefore be worthwhile even if the system is only being used by one user.

makesys[file] — creates a saved file in which all pages in this system, including private user pages, are made read execute, i.e. shared. This system can then be run via the TENEX command RUN, or GET and START.

For example, new INTERLISP systems are brought up by loading the appropriate compiled files and then performing makesys[LISP.SAV].[17]

---

[17] makesys is also advised to set the variable makesysdate to (DATE), i.e. the time and date the system was made.

Index for Section 3

Page Numbers

SECTION 4

FUNCTION TYPES AND IMPLICIT PROGN

In INTERLISP, each function may independently have:

a. its arguments evaluated or not evaluated;

b. a fixed number of arguments or an indefinite number of arguments;

c. be defined by an INTERLISP expression, by built-in machine code, or by compiled machine code.

Hence there are twelve function types (2 x 2 x 3).

4.1 Exprs

Functions defined by INTERLISP expressions are called exprs. Exprs must begin with either LAMBDA or NLAMBDA,[1] indicating whether the arguments to the function are to be evaluated or not evaluated, respectively. Following the LAMBDA or NLAMBDA in the expr is the 'argument list', which is either

(1) a list of literal atoms or NIL (fixed number of arguments); or

(2) any literal atom other than NIL, (indefinite number of arguments).

Case (1) corresponds to a function with a *fixed* number of arguments. Each atom in the list is the *name* of an argument for the function defined by this

---

[1] Where unambiguous, the term expr is used to refer to either the function, or its definition.

expression. When the function is called, its arguments will be evaluated or not evaluated, as dictated by whether the definition begins with LAMBDA or NLAMBDA, and then paired with these argument names.[2] This process is called "spreading" the arguments, and the function is called a spread-LAMBDA or a spread-NLAMBDA.

Case (2) corresponds to a function with an *indefinite* number of arguments. Such a function is called a <u>nospread</u> function. If its definition begins with NLAMBDA, the atom which constitutes its argument list is bound to the list of arguments to the function (unevaluated). For example, if FOO is defined by (NLAMBDA X --), when (FOO THIS IS A TEST) is evaluated, X will be bound to (THIS IS A TEST).

If a <u>nospread</u> function begins with a LAMBDA, indicating its arguments are to be evaluated, each of its <u>n</u> arguments are evaluated and their values stored on the pushdown list. The atom following the LAMBDA is then bound to the *number* of arguments which have been evaluated. For example, if FOO is defined by (LAMBDA X --) when (FOO A B C) is evaluated, A, B, and C are evaluated and X is bound to 3. A built-in function, arg[atm;m], is available for computing the value of the <u>m</u>th argument for the lambda-atom variable <u>atm</u>. <u>arg</u> is described in section 8.

4.2 Compiled Functions

Functions defined by expressions can be compiled by the INTERLISP compiler, as

---

[2] Note that the function itself can evaluate selected arguments by calling <u>eval</u>. In fact, since the function type can specify only that all arguments are to be evaluated or none are to be evaluated, if it is desirable to write a function which only evaluates *some* of its arguments, e.g. <u>setq</u>, the function is defined as an nlambda, i.e. no arguments are evaluated in the process of calling the function, and then included in the definition itself are the appropriate calls to <u>eval</u>.

described in section 18, "The Compiler and Assembler". Functions may also be written directly in machine code using the ASSEMBLE directive of the compiler. Functions created by the compiler, whether from S-expressions or ASSEMBLE directives, are referred to as compiled functions.

4.3 Function Type

The function fntyp returns the function type of its argument. The value of fntyp is one of the following 12 types:

EXPR	CEXPR	SUBR
FEXPR	CFEXPR	FSUBR
EXPR*	CEXPR*	SUBR*
FEXPR*	CFEXPR*	FSUBR*

The types in the first column are all defined by expressions. The types in the second column are compiled versions of the types in the first column, as indicated by the prefix C. In the third column are the parallel types for built-in subroutines. Functions of types in the first two rows have a fixed number of arguments, i.e., are spread functions. Functions in the third and fourth rows have an indefinite number of arguments, as indicated by the suffix *. The prefix F indicates no evaluation of arguments. Thus, for example, a CFEXPR* is a compiled form of a nospread-NLAMBDA.

A standard feature of the INTERLISP system is that no error occurs if a spread function is called with too many or too few arguments. If a function is called with too many arguments, the extra arguments are evaluated but ignored. If a function is called with too few arguments, the unsupplied ones will be delivered as NIL. In fact, the function itself cannot distinguish between being given NIL as an argument, and not being given that argument, e.g., (FOO) and (FOO NIL) are exactly the same for spread functions.

4.4 Progn

progn is a function of an arbitrary number of arguments. progn evaluates the arguments in order and returns the value of the last, i.e., it is an extension of the function prog2 of LISP 1.5. Both cond and lambda/nlambda expressions have been generalized to permit 'implicit progns' as described below.

4.5 Implicit Progn

The conditional expression has been generalized so that each clause may contain n forms ($n \geq 1$) which are interpreted as follows:

```
(COND
   (P1 E11 E12 E13)
   (P2 E21 E22)                    [1]
   (P3)
   (P4 E41))
```

will be taken as equivalent to (in LISP 1.5):

```
(COND
   (P1 (PROGN E11 E12 E13))
   (P2 (PROGN E21 E22))
   (P3 P3)                         [2]
   (P4 E41)
   (T NIL))
```

Note however that P3 is evaluated only once in [1], while it is evaluated a second time if the expression is written as in [2]. Thus a clause in a cond with only a predicate and no following expression causes the value of the predicate itself, if non-NIL, to be returned. Note also that NIL is returned if all the predicates have value NIL, i.e., the cond 'falls off the end'. No error is generated.

LAMBDA and NLAMBDA expressions also allow implicit progn's; thus for example:

```
(LAMBDA (V1 V2) (F1 V1) (F2 V2) NIL)
```

is interpreted as:

```
(LAMBDA (V1 V2) (PROGN (F1 V1) (F2 V2) NIL))
```

The value of the last expression following LAMBDA (or NLAMBDA) is returned as the value of the entire expression. In this example, the function would always return NIL.

Index for Section 4

SECTION 5

PRIMITIVE FUNCTIONS AND PREDICATES

5.1 Primitive Functions

car[x]

car gives the first element of a list x, or the left element of a dotted pair x. For literal atom, value is top level binding (value) of the atom. For all other nonlists, e.g. strings, arrays, and numbers, the value is undefined, i.e., it is the right 18 bits of x.

cdr[x]

cdr gives the rest of a list (all but the first element). This is also the right member of a dotted pair. If x is a literal atom, cdr[x] gives the property list of x. Property lists are usually NIL unless modified by the user. The value of cdr is undefined for other nonlists, i.e. it is the left 18 bits of x.

caar[x] = car[car[x]]
cadr[x] = car[cdr[x]]
cddddr[x] =
cdr[cdr[cdr[cdr[x]]]]

All 30 combinations of nested cars and cdrs up to 4 deep are included in the system. All are compiled open by the compiler.

cons[x;y]

cons constructs a dotted pair of x and y. If y is a list, x becomes the first element of that list.

To minimize drum accesses the following algorithm is used for finding a page on which to put the constructed INTERLISP word.

cons[x;y] is placed

1) on the page with y if y is a list and there is room; otherwise
2) on the page with x if x is a list and there is room; otherwise
3) on the same page as the last cons if there is room; otherwise
4) on any page *with a specified minimum* of storage, presently 16 LISP words.

conscount[] — value is the number of conses since this INTERLISP was started up.

rplacd[x;y] — Places the pointer y in the decrement, i.e. cdr, of the cell pointed to by x. Thus it physically changes the internal list structure of x, as opposed to cons which creates a new list element. The only way to get a circular list is by using rplacd to place a pointer to the beginning of a list in a spot at the end of the list.

The value of rplacd is x. An attempt to rplacd NIL will cause an error, ATTEMPT TO RPLAC NIL, (except for rplacd[NIL;NIL]). For x a literal atom, rplacd[x;y] will make y be the property list of x. For all other non-lists, rplacd should be used with care: it will simply store y in the left 18 bits of x.

rplaca[x;y] similar to <u>rplacd</u>, but replaces the address pointer of <u>x</u>, i.e., <u>car</u>, with <u>y</u>. The value of <u>rplaca</u> is <u>x</u>. An attempt to <u>rplaca</u> NIL will cause an error, ATTEMPT TO RPLAC NIL, (except for rplaca[NIL;NIL]). For <u>x</u> a literal atom, rplaca[x;y] will make <u>y</u> be the top level value for <u>x</u>. For all other non-lists, <u>rplaca</u> should be used with care: it will simply store <u>y</u> in the right 18 bits of <u>x</u>.

Convention: Naming a function by prefixing an existing function name with <u>f</u> usually indicates that the new function is a <u>f</u>ast version of the old, i.e., one which has the same definition but compiles open and runs without any 'safety' error checks.

frplacd[x;y] Has the same definition as <u>rplacd</u> but compiles open as one instruction. Note that no checks are made on <u>x</u>, so that a compiled <u>frplacd</u> can clobber NIL, producing strange and wondrous effects.

frplaca[x;y] Similar to <u>frplacd</u>.

quote[x] This is a function that prevents its arguments from being evaluated. Its value is <u>x</u> itself, e.g. (QUOTE FOO) is FOO.[1]

kwote[x] (LIST (QUOTE QUOTE) x),
if <u>x</u>=A, and <u>y</u>=B, then
(KWOTE (CONS x y))= (QUOTE (A . B)).

---

[1] Since giving <u>quote</u> more than one argument, e.g. (QUOTE EXPR (CONS X Y)), is almost always a parentheses error, and one that would otherwise go undetected, <u>quote</u> itself generates an error in this case, PARENTHESIS ERROR.

$cond[c_1;c_2;\ldots;c_k]$

The conditional function of INTERLISP, cond, takes an indefinite number of arguments $c_1, c_2, \ldots c_k$, called clauses. Each clause c_i is a list ($e_{1i} \ldots e_{ni}$) of $n \geq 1$ items, where the first element is the predicate, and the rest of the elements the consequents. The operation of cond can be paraphrased as IF e_{11} THEN $e_{21} \ldots e_{n1}$ ELSEIF e_{12} THEN $e_{22} \ldots e_{n2}$ ELSEIF $e_{13} \ldots$

The clauses are considered in sequence as follows: the first expression e_{1i} of the clause c_i is evaluated and its value is classified as false (equal to NIL) or true (not equal to NIL). If the value of e_{1i} is true, the expressions $e_{2i} \ldots e_{ni}$ that follow in clause c_i are evaluated in sequence, and the value of the conditional is the value of e_{ni}, the last expression in the clause. In particular, if $n=1$, i.e., if there is only one expression in the clause c_i, the value of the conditional is the value of e_{1i}. (which is evaluated only once).

If e_{1i} is false, then the remainder of clause c_i is ignored, and the next clause c_{i+1} is considered. If no e_{1i} is true for any clause, the value of the conditional expression is NIL.

$selectq[x;y_1;y_2;\ldots;y_n;z]$

selects a form or sequence of forms based on the value of its first argument x. Each y_i is a list of the form (s_i e_{1i} e_{2i} $\ldots$ e_{ki}) where s_i is the selection key. The operation of selectq can be paraphrased as:

IF $\underline{x}=s_1$ THEN e_{1i} ... e_{ki}
ELSEIF $\underline{x}=s_2$ THEN ... ELSE z.

If $\underline{s}_i$ is an atom, the value of $\underline{x}$ is tested to see if it is eq to $\underline{s}_i$ (not evaluated). If so, the expressions e_{1i} ... e_{ki} are evaluated in sequence, and the value of the selectq is the value of the last expression evaluated, i.e. $\underline{e}_{ki}$.

If $\underline{s}_i$ is a list, the value of $\underline{x}$ is compared with each element (not evaluated) of $\underline{s}_i$, and if $\underline{x}$ is eq to any one of them, then e_{1i} to e_{ki} are evaluated in turn as above.

If $\underline{y}_i$ is not selected in one of the two ways described, $\underline{y}_{i+1}$ is tested, etc., until all the y's have been tested. If none is selected, the value of the selectq is the value of z. z must be present.

An example of the form of a selectq is:

```
[SELECTQ (CAR X)
         (Q (PRINT FOO)
            (FIE X))
         ((A E I O U)
           (VOWEL X))
         (COND
           ((NULL X)
             NIL)
           (T (QUOTE STOP]
```

which has two cases, Q and (A E I O U) and a default condition which is a cond.

selectq compiles open, and is therefore very fast;

however, it will not work if the value of x is a list, a large integer, or floating point number, since selectq uses eq for all comparisons.

prog1[x_1;x_2;...;x_n]

evaluates its arguments in order, that is, first x_1, then x_2, etc, and returns the value of its first argument x_1, e.g. (PROG1 X (SETQ X Y)) sets x to y, and returns x's original value.

progn[x_{1i};x_{2i};...;x_n]

progn evaluates each of its arguments in order, and returns the value of its last argument as its value. progn is used to specify more than one computation where the syntax allows only one, e.g. (SELECTQ ... (PROGN ...)) allows evaluation of several expressions as the default condition for a selectq.

prog[args;e_1;e_2;...;e_n]

This function allows the user to write an ALGOL-like program containing INTERLISP expressions (forms) to be executed. The first argument, args, is a list of local variables (must be NIL if no variables are used). Each atom in args is treated as the name of a local variable and bound to NIL. args can also contain lists of the form (atom form). In this case, atom is the name of the variable and is bound to the value of form. The evaluation takes place before any of the bindings are performed, e.g., (PROG ((X Y) (Y X)) ...) will bind x to the value of y and y to the (original) value of x.

The rest of the prog is a sequence of non-atomic statements (forms) and atomic symbols used as labels for go. The forms are evaluated sequentially; the labels serve only as markers. The two special functions go and return alter this flow of control as described below. The value of the prog is usually specified by the function return. If no return is executed, i.e., if the prog "falls off the end," the value of the prog is undefined, i.e. garbage.

go[x]

go is the function used to cause a transfer in a prog. (GO L) will cause the program to continue at the label L. A go can be used at any level in a prog. If the label is not found, go will search higher progs *within the same function*, e.g. (PROG -- A -- (PROG -- (GO A))). If the label is not found in the function in which the prog appears, an error is generated, UNDEFINED OR ILLEGAL GO.

return[x]

A return is the normal exit for a prog. Its argument is evaluated and is the value of the prog in which it appears.

If a go or return is executed in an interpreted function which is not a prog, the go or return will be executed in the last interpreted prog entered if any, otherwise cause an error.

go or return inside of a compiled function that is not a prog is not allowed, and will cause an error at compile time.

As a corollary, go or return in a functional argument, e.g. to mapc, will not

work compiled. Also, since nlsetq's and ersetq's compile as *separate* functions, a go or return *cannot* be used inside of a compiled nlsetq or ersetq if the corresponding prog is outside, i.e. above, the nlsetq or ersetq.

set[x;y] This function sets x to y. Its value is y. If x is not a literal atom, causes an error, ARG NOT ATOM - SET. If x is NIL, causes an error, ATTEMPT TO SET NIL. Note that set is a normal lambda-spread function, i.e., its arguments are evaluated before it is called. Thus, if the value of x is c, and the value of y is b, then set[x;y] would result in c having value b, and b being returned as the value of set.

setq[x;y] An nlambda version of set: the first argument is not evaluated, the second is.[2] Thus if the value of X is C and the value of Y is B, (SETQ X Y) would result in X (not C) being set to B, and B being returned. If x is not a literal atom, an error is generated, ARG NOT ATOM - SET. If x is NIL, the error ATTEMPT TO SET NIL is generated.

setqq[x;y] Like setq except that neither argument is evaluated, e.g. (SETQQ X (A B C)) sets x to (A B C).

---

2 Since setq is an nlambda, *neither* argument is evaluated during the calling process. However, setq itself calls eval on its second argument. Note that as a result, typing (SETQ var form) and SETQ(var form) to lispx is equivalent: in both cases var is not evaluated, and form is.

rpaq[x;y] — like setq, except always works on top level binding of x, i.e. on the value cell. rpaq derives its name from rplaca quote, since it is essentially an nlambda version of rplaca, e.g. (RPAQ FOO form) is equivalent to (RPLACA (QUOTE FOO) form).

rpaqq[x;y] — like setqq for top level bindings.

rpaq and rpaqq are used by prettydef (Section 14). Both rpaq and rpaqq generate errors if x is not atomic. Both are affected by the value of dfnflg (Section 8). If dfnflg = ALLPROP (and the value of x is other than NOBIND), instead of setting x, the corresponding value is stored on the property list of x under the property VALUE.

Resetvar and Resetform

resetvar[var;new-value;from] — The effect of resetvar is the same as (PROG ((var new-value)) (RETURN form)), except that resetvar is designed to work on GLOBAL variables, i.e. variables that must be reset, not rebound (see section 18). resetvar resets the variable (using frplaca), and then restores its value after evaluating form. The evaluation of form is errorset protected so that the value is restored even if an error occurs.[3] resetvar also adds the old value of var to a global list, so

---

[3] In this case, after restoring the value, resetvar propagates the error backwards by calling error!.

that if the user types Control-D (or Control-C followed by REENTER) while *form* is being evaluated, the variable will be restored by the top level INTERLISP executive. The value of *resetvar* is the value returned by *form*. *resetvar* compiles open.

For example, the editor calls *lispx* to execute editor history commands by performing (RESETVAR LISPXHISTORY EDITHISTORY (LISPX --)), thereby making *lispx* work on *edithistory* instead of *lispxhistory*.

The behavior of many system functions is affected by calling certain functions, as opposed to resetting variables, e.g. *printlevel*, *linelength*, *input*, *output*, *radix*, *gcgag*, etc. The function *resetform* enables a program to treat these functions much like variables, and temporarily change their "setting".

resetform[form1;form2]

nlambda, nospread. *form1* is evaluated, then *form2* is evaluated, then *form1* is 'restored', e.g. (RESETFORM (RADIX 8) (FOO)) will evaluate (FOO) while *radix* is 8, and then restore the original setting of *radix*.

form1 must return as its value its "previous setting" so that its effects can be undone by applying *car* of *form1* to this value.

resetform is errorset protected like *resetvar*, and also records its information on a global list so that after control-D (or control-C REENTER), *form1* is properly restored.

The value of resetform is the value returned by form2. resetform compiles open.

5.2 Predicates and Logical Connectives

atom[x] — is T if x is an atom; NIL otherwise.

litatom[x] — is T if x is a literal atom, i.e., an atom and not a number, NIL otherwise.

numberp[x] — is x if x is a number, NIL otherwise.

Convention: Functions that end in p are usually predicates, i.e. they test for some condition.

stringp[x] — is x if x is a string, NIL otherwise.[4]

arrayp[x] — is x if x is an array, NIL otherwise.

listp[x] — is x if x is a list-structure, i.e., one created by one or more conses; NIL otherwise.

Note that arrays and strings are not atoms, but are also not lists, i.e. both atom and listp will return NIL when given an array or a string.

nlistp[x] — not[listp[x]]

eq[x;y] — The value of eq is T, if x and y are pointers to

---

[4] For other string functions, see Section 10.

the same structure in memory, and NIL otherwise. eq is compiled open by the compiler as a 36 bit compare of pointers. Its value is not guaranteed T for equal numbers which are not small integers. See eqp.

neq[x;y] — The value of neq is T, if x is not eq to y, and NIL otherwise.

null[x] — eq[x;NIL]

not[x] — same as null, that is eq[x;NIL].

eqp[x;y] — The value of eqp is T if x and y are eq, i.e. pointers to the same structure in memory, or if x and y are numbers and are equal. Its value is NIL otherwise.[6]

equal[x;y] — The value of this function is T if x and y print identically; the value of equal is NIL otherwise. Note that x and y do *not* have to be eq.

and[x_1;x_2;...;x_n] — Takes an indefinite number of arguments (including 0). If all of its arguments have non-null value, its value is the value of its last argument, otherwise NIL. E.g. and[x;member[x;y]] will have as its value either NIL or a tail of y. and[]=T. Evaluation stops at the first argument whose value is NIL.

---

[6] For other number functions, see Section 13.

$or[x_1;x_2;...;x_n]$

Takes an indefinite number of arguments (including 0). Its value is that of the first argument whose value is not NIL, otherwise NIL if all arguments have value NIL. E.g. or[x;numberp[y]] has its value x, y, or NIL. or[]=NIL. Evaluation stops at the first argument whose value is not NIL.

every[everyx;everyfn1;everyfn2]

Is T if the result of applying everyfn1 to each element in everyx is true, otherwise NIL. E.g., every[(X Y Z); ATOM]=T.

every operates by computing everyfn1[car[everyx]].[6] If this yields NIL, every immediately returns NIL. Otherwise, every computes everyfn2[everyx], or cdr[everyx] if everyfn2=NIL, and uses this as the 'new' everyx, and the process continues, e.g. every[x;ATOM;CDDR] is true if every *other* element of x is atomic.

every compiles open.

some[somex;somefn1;somefn2]

value is the tail of somex beginning with the first element that satisfies somefn1, i.e., for which somefn1 applied to that element is true. Value is NIL if no such element exists. E.g., some[x;(LAMBDA (Z) (EQUAL Z Y))] is equivalent to member[y;x]. some operates

---

[6] Actually, everyfn1[car[everyx];everyx] is computed, so for example everyfn1 can look at the next element on everyx if necessary.

analagously to every. At each stage, somefn1[car[somex];somex] is computed, and if this is not NIL, somex is returned as the value of some. Otherwise, somefn2[somex] is computed, or cdr[somex] if somefn2=NIL, and used for the next somex.

some compiles open.

notany[somex;somefn1,somefn2] same as not[some[somex;somefn1;somefn2]]

notevery[everyx;everyfn1;everyfn2] not[every[everyx;everyfn1;everyfn2]]

memb[x;y] Determines if x is a member of the list y, i.e., if there is an element of y eq to x. If so, its value is the tail of the list y starting with that element. If not, its value is NIL.

fmemb[x;y] Fast version of memb that compiles open as a five instruction loop, terminating on a NULL check. Interpreted, fmemb gives an error, BAD ARGUMENT - FMEMB, if y ends in a non-list other than NIL.

member[x;y] Identical to memb except that it uses equal instead of eq to check membership of x in y.

The reason for the existence of both memb and member is that eq compiles as one instruction but equal requires a function call, and is therefore considerably more expensive. Wherever possible, the user should write (and use) functions that use eq instead of equal.

tailp[x;y] — Is x, if x is a list and a *tail* of y, i.e., x is eq to some number of cdrs $\geq$ 0[7] of y, NIL otherwise.

assoc[x;y] — y is a list of lists (usually dotted pairs). The value of assoc is the first sublist of y whose car is eq to x. If such a list is not found, the value is NIL. Example:
assoc[B;((A . 1) (B . 2) (C . 3))] = (B . 2).

fassoc[x;y] — Fast version of assoc that compiles open as a 6 instruction loop, terminating on a NULL check. Interpreted, fassoc gives an error if y ends in a non-list other than NIL, BAD ARGUMENT - FASSOC.

sassoc[x;y] — Same as assoc but uses equal instead of eq.

---

[7] If x is eq to some number of cdrs $\geq$ 1 of y, we say x is a proper tail.

Index for Section 5

Page Numbers

SECTION 6

LIST MANIPULATION AND CONCATENATION

list[x_1;x_2;...;x_n]

lambda-nospread function. Its value is a list of the values of its arguments.

append[x_1;x_2;...;x_n]

Copies the top level of the list $\underline{x}_1$ and appends this to a copy of top level list $\underline{x}_2$ appended to ... appended to $\underline{x}_n$, e.g.
append[(A B) (C D E) (F G)] = (A B C D E F G).
Note that only the first n-1 lists are copied. However n=1 is treated specially; i.e. append[x] can be used to copy the top level of a single list.[1]

The following examples illustrate the treatment of non-lists.

append[(A B C);D] = (A B C . D)
append[A;(B C D)] = (B C D)
append[(A B C . D);(E F G)] = (A B C E F G)
append[(A B C . D)] = (A B C . D)

---

[1] To copy a list to all levels, use <u>copy</u>.

$nconc[x_1;x_2;...;x_n]$

Returns same value as <u>append</u> but actually modifies the list structure of $x_1 \ldots x_{n-1}$.

nconc1[lst;x]

Performs nconc[lst;list[x]]. The <u>cons</u> will be on the same page as <u>lst</u>.

tconc[ptr;x]

<u>tconc</u> is useful for building a list by adding elements one at a time at the end, i.e. its role is similar to that of <u>nconc1</u>. However, unlike <u>nconc1</u>, <u>tconc</u> does not have to search to the end of the list each time it is called. It does this by keeping a pointer to the end of the list being assembled, and updating this pointer after each call. The savings can be considerable for long lists. The cost is the extra word required for storing both the list being assembled, and the end of the list. <u>ptr</u> is that word: car[ptr] is the list being assembled, cdr[ptr] is last [car[ptr]]. The value of <u>tconc</u> is <u>ptr</u>, with the appropriate modifications to <u>car</u> and <u>cdr</u>. Example:

```
←(RPTQ 5 (SETQ FOO TCONC FOO RPTN)))
((5 4 3 2 1) 1)
```

<u>tconc</u> can be initialized in two ways. If <u>ptr</u> is NIL, <u>tconc</u> will make up a <u>ptr</u>. In this case, the program must set some variable to the value of the first call to <u>tconc</u>. After that, it is unnecessary to reset <u>ptr</u> since <u>tconc</u> physically changes it. Thus:

```
←(SET FOO (TCONC NIL 1))
((1) 1)
←(RPTQ 4 (TCONC FOO RPTN))
((1 4 3 2 1) 1)
```

If ptr is initially (NIL), the value of tconc is the same as for ptr=NIL, but tconc changes ptr, e.g.

```
←(SETQ FOO (CONS))
(NIL)
←(RPTQ 5 (TCONC FOO RPTN))
((5 4 3 2 1) 1)
```

The latter method allows the program to initialize, and then call tconc without having to perform setq on its value.

lconc[ptr;x]

Where tconc is used to add *elements* at the end of a list, lconc is used for building a list by adding *lists* at the end, i.e. it is similar to nconc instead of nconc1, e.g.

```
←(SETQ FOO (CONS))
(NIL)
←(LCONC FOO (LIST 1 2))
((1 2) 2)
←(LCONC FOO (LIST 3 4 5))
((1 2 3 4 5) 5)
←(LCONC FOO NIL)
((1 2 3 4 5) 5)
```

Note that

```
←(TCONC) FOO NIL)
((1 2 3 4 5 NIL) NIL)
←(TCONC FOO (LIST 3 4 5))
((1 2 3 4 5 NIL (3 4 5)) (3 4 5))
```

lconc uses the same pointer conventions as tconc for eliminating searching to the end of the list, so that the same pointer can be given to tconc and lconc interchangeably.

attach[x;y]

Value is equal to cons[x;y], but attaches x to the front of y by doing an rplaca and rplacd, i.e. the value of attach is eq to y, which it

physically changes. y must be a list, or an error is generated, ILLEGAL ARG.

remove[x;l]

Removes all occurrences of x from list l, giving a copy of l with all elements equal to x removed.

Convention: Naming a function by prefixing an existing function with d frequently indicates the new function is a destructive version of the old one, i.e. it does not make any new structure but cannibalizes its argument(s).

dremove[x;l]

Similar to remove, but uses eq instead of equal, and actually modifies the list l when removing x, and thus does not use any additional storage. More efficient than remove.

copy[x]

Makes a copy of the list x. The value of copy is the copied list. All levels of x are copied,[2] down to non-lists, so that if x contains arrays and strings, the copy of x will contain the same arrays and strings, not copies. Copy is recursive in the car direction only, so that very long lists can be copied.

reverse[l]

Reverses (and copies) the top level of a list, e.g. reverse[(A B (C D))] = ((C D) B A). If x is not a list, value is x.

dreverse[l]

Value is same as that of reverse, but dreverse

---

[2] To copy just the *top* level of x, do append[x].

destroys the original list l and thus does not use any additional storage. More efficient than reverse.

subst[x;y;z]

Value is the result of substituting the S-expression x for all occurrences of the S-expression y in the S-expression z. Substitution occurs whenever y is equal to car of some subexpression of z, or when y is both atomic and not NIL and eq to cdr of some subexpression of z. For example:

```
subst[A;B;(C B (X . B))] = (C A (X . A))
subst[A;(B C);((B C) D B C)] = (A D B C),
      not (A D . A).
```

The value of subst is a copy of z with the appropriate changes. Furthermore, if x is a list, it is copied at each substitution.

dsubst[x;y;z]

Similar to subst, but uses eq and does not copy z, but changes the list structure z itself. Like subst, dsubst substitutes with a copy of x. More efficient than subst.

lsubst[x;y;z]

Like subst except x is substituted as a segment, e.g. lsubst[(A B);Y;(X Y Z)] is (X A B Z). Note that if x is NIL, produces a copy of z with all y's deleted.

esubst[x;y;z;flg]

Similar to dsubst, but first checks to see if y actually appears in z. If not, calls error! where

flg=T means print a message of the form x ? This function is actually an implementation of the editor's R command (see Section 9), so that y can use &, --, or alt-modes as with the R command.

sublis[alst;expr;flg]

alst is a list of pairs:
$((u_1 \cdot v_1)\ (u_2 \cdot v_2) \ldots (u_n \cdot v_n))$ with each u_i atomic.

The value of sublis[alst;expr;flg] is the result of substituting each v for the corresponding u in expr.[3] Example:
sublis[((A . X) (C . Y));(A B C D)] = (X B Y D)
New structure is created only if needed, or if flg=T, e.g. if flg=NIL and there are no substitutions, value is eq to expr.

subpair[old;new;expr;flg]

Similar to sublis, except that elements of new are substituted for corresponding atoms of old in expr. Example:
subpair[(A C);(X Y);(A B C D)] = (X B Y D)
As with sublis, new structure is created only if needed, or if flg=T, e.g. if flg=NIL and there are no substitutions, the value is eq to expr.

If old ends in an atom other than NIL, the rest of the elements on new are substituted for that atom. For example, if old=(A B . C) and new=(U V X Y Z),

---

[3] To remember the order on alst, think of it as old to new, i.e. u_i -> v_i.

U is substituted for A, V for B, and (X Y Z) for C. Similarly, if old itself is an atom (other than NIL), the entire list new is substituted for it.

Note that subst, dsubst, lsubst, and esubst all substitute copies of the appropriate expression, whereas subpair and sublis substitute the identical structure (unless flg=T).

last[x]

Value is a pointer to the last *node* in the list x, e.g. if x=(A B C) then last[x] = (C). If x=(A B . C) last[x] = (B . C). Value is NIL if x is not a list.

flast[x]

Fast version of last that compiles open as a 5 instruction loop, terminating on a null-check. Interpreted, generates an error, BAD ARGUMENT - FLAST, if x ends in other than NIL.

nleft[l;n;tail]

Tail is a tail of l or NIL. The value of nleft is the tail of l that contains n more elements than tail,[4] e.g., if x=(A B C D E), nleft[x;2]=(D E), nleft[x;1;cddr[x]]=(B C D E). Thus nleft can be used to work backwards through a list. Value is NIL if l does not contain n more elements than tail.

lastn[l;n]

Value is cons[x;y] where y is the last n elements of l, and x is the initial segment, e.g. lastn[(A B C D E);2]=((A B C) D E)

---

[4] If tail is not NIL, but not a tail of l, the result is the same as if tail were NIL, i.e. nleft operates by scanning l looking for tail, not by computing the lengths of l and tail.

lastn[(A B);2]=(NIL A B).

Value is NIL if l is not a list containing at least n elements.

nth[x;n]

Value is the tail of x beginning with the nth element, e.g. if n=2, value is cdr[x], if n=3, cddr[x], etc. If n=1, value is x, if n=0, for consistency, value is cons[NIL;x]. If x has fewer than n elements, value is NIL, e.g. nth[(A B);3]=NIL, as is nth[(A . B);3] Note that nth[(A . B);2]=B.

fnth[x;n]

Fast version of nth that compiles open as a 3 instruction loop, terminating on a null-check. Interpreted, generates an error, BAD ARGUMENT - FNTH, if x ends in other than NIL.

length[x]

Value is the length of the list x where length is defined as the number of cdrs required to reach a non-list, e.g.

length[(A B C)] = 3

length[(A B C . D)] = 3

length[A] = 0

flength[x]

Fast version of length that compiles open as a 4 instruction loop, terminating on a null-check. Interpreted, generates an error, BAD ARGUMENT - FLENGTH, if x ends in other than NIL.

count[x]

Value is the number of list words in the structure

x. Thus, count is like a length that goes to all levels. Count of a non-list is 0.

ldiff[x;y;z]

y must be a tail of x, i.e. eq to the result of applying some number of cdrs to x. ldiff[x;y] gives a list of all elements in x up to y, i.e., the list difference of x and y. Thus ldiff[x;member[FOO;x]] gives all elements in x up to the first FOO.

Note that the value of ldiff is always new list structure unless y=NIL, in which case the value is x itself.

If z is not NIL the value of ldiff is effectively nconc[z;ldiff[x;y]], i.e. the list difference is added at the end of z.

If y is not a tail of x, generates an error, LDIFF: NOT A TAIL. ldiff terminates on a null-check.

intersection[x;y]

Value is a list whose elements are members of both lists x and y. Note that intersection[x;x] gives a list of all members of x without any duplications.

union[x;y]

Value is a (new) list consisting of all elements included on either of the two original lists. It

is more efficient to make x be the shorter list.[5]

sort[data;comparefn][6]

data is a list of items to be sorted using comparefn, a predicate function of two arguments which can compare any two items on data and return T if the first one belongs before the second. If comparefn is NIL, alphorder is used; thus sort[data] will alphabetize a list. If comparefn is T, car's of items are given to alphorder; thus sort[a-list;T] will alphabetize by the car of each item. sort[x;ILESSP] will sort a list of integers.

The value of sort is the sorted list. The sort is destructive and uses no extra storage. The value returned is eq to data but elements have been switched around. Interrupting with control D, E, or B may cause loss of data, but control H may be used at any time, and sort will break at a clean state from which ↑ or control characters are safe. The algorithm has been optimized with respect to the number of compares.

Note: if comparefn[a;b] = comparefn[b;a], then the ordering of a and b may or may not be preserved.

---

5 The value of union is y with all elements of x not in y consed on the front of it. Therefore, if an element appears twice in y, it will appear twice in union[x;y]. Also, since union[(A);(A A)] = (A A), while union[(A A);(A)] = (A), union is non-commutative.

6 Sort, merge, and alphorder were written by J.W. Goodwin.

For example, if (FOO . FIE) appears before (FOO . FUM) in x, sort[x;T] may or may not reverse the order of these two elements. Of course, the user can always specify a more precise comparefn.

merge[a;b;comparefn]

a and b are lists which have previously been sorted using sort and comparefn. Value is a destructive merging of the two lists. It does not matter which list is longer. After merging both a and b are equal to the merged list. (In fact, cdr[a) is eq to cdr[b]) merge may be aborted after control H.

alphorder[a;b]

A predicate function of two arguments, for alphabetizing. Returns T if its arguments are in order, i.e. if b does not belong before a. Numbers come before literal atoms, and are ordered by magnitude (using greaterp). Literal atoms and strings are ordered by comparing the (ASCII) character codes in their pnames. Thus alphorder[23;123] is T, whereas alphorder[A23;A123] is NIL, because the character code for the digit 2 is greater than the code for 1.

Atoms and strings are ordered before all other data types. If neither a nor b are atoms or strings, the value of alphorder is T, i.e. in order.

Note: alphorder does no unpacks, chcons, conses or nthchars. It is several times faster for alphabetizing than anything that can be written using these other functions.

cplists[x;y] compares x and y and prints their differences, i.e. cplists is essentially a SRCCOM for list structures.

Index for Section 6

SECTION 7

PROPERTY LISTS AND HASH LINKS

7.1 Property Lists

Property lists are entities associated with literal atoms, and are stored on cdr of the atom. Property lists are conventionally lists of the form (property value property value ... property value) although the user can store anything he wishes in cdr of a literal atom. However, the functions which manipulate property lists observe this convention by cycling down the property lists two cdrs at a time. Most of these functions also generate an error, ARG NOT ATOM, if given an argument which is not a literal atom, i.e., they cannot be used directly on lists.

The term 'property name' or 'property' is used for the property indicators appearing in the odd positions, and the term 'property value' or 'value of a property' or simply 'value' for the values appearing in the even positions. Sometimes the phrase 'to store on the property --' is used, meaning to place the indicated information on the property list under the property name --.

Properties are usually atoms, although no checks are made to eliminate use of non-atoms in an odd position. However, the property list searching functions all use eq.

Property List Functions

put[atm;prop;val] puts on the property list of atm, the property prop with value val. val replaces any previous value for the property prop on this property list. Generates an error, ARG NOT ATOM, if atm is not a literal atom. Value is val.

addprop[atm;prop;new;flg] adds the value new to the list which is the value of property prop on property list of atm. If flg is T, new is consed onto the front of value of

prop, otherwise it is nconced on the end (nconc1). If atm does not have a property prop, the effect is the same as put[atm;prop;list[new]], for example, if addprop[FOO;PROP;FIE] is followed by addprop[FOO;PROP;FUM], getp[FOO;PROP] will be (FIE FUM). The value of addprop is the (new) property value. If atm is not a literal atom, generates an error, ARG NOT ATOM.

remprop[atm;prop]

removes all occurrences of the property prop (and its value) from the property list of atm. Value is prop if any were found, otherwise NIL. If atm is not a literal atom, generates an error, ARG NOT ATOM.

changeprop[x;prop1;prop2]

Changes *name* of property prop1 to prop2 on property list of x, (but does not affect the value of the property). Value is x, unless prop1 is not found, in which case, the value is NIL. If x is not a literal atom, generates an error, ARG NOT ATOM.

get[x;y]

Gets the item after the atom y on list x. If y is not on the list x, value is NIL. For example, get[A B C D);B]=C.

Note: since get terminates on a non-list, get[atom;anything] is NIL.

Therefore, to search a property list, getp should be used, or get applied to cdr[atom].

getp[atm;prop]

gets the property value for prop from the property list of atm. The value of getp is NIL if atm is not a literal atom, or prop if not found.

Note: the value of getp may also be NIL, if there is an occurrence of prop but the corresponding property value is NIL.

Note: Since getp searches a list two items at a time, the same object can be used as both a property name and a property value, e.g., if the property list of atm is (PROP1 A PROP2 B A C), then getp[atm;A] = C. Note however that get[cdr[atm];A] = PROP2.

getlis[x;props]

searches the property list of x, and returns the property list as of the first property on props that it finds e.g., if the property list of x is (PROP1 A PROP3 B A C),
getlis[x;(PROP2 PROP3)]=(PROP3 B A C)
Value is NIL if no element on props is found. x can also be a list itself, in which case it is searched as above.

deflist[l;prop]

is used to put values under the same property name on the property lists of several atoms. l is a list of two-element lists. The first element of each is a literal atom, and the second element is the property value for the property prop. The value of deflist is NIL.

Note: Many atoms in the system already have property lists, with properties used by the compiler, the break package, DWIM, etc. Be careful not to clobber such system properties. The value of sysprops gives the complete list of the property names used by the system.

7.2 Hash Links

The description of the hash link facility in INTERLISP is included in the chapter on property lists because of the similarities in the ways the two features are used. A property list provides a way of associating information with a particular atom. A hash link is an association between any INTERLISP pointer (atoms, numbers, arrays, strings, lists, et al) called the hash-item, and any other INTERLISP pointer called the hash-value. Property lists are stored in cdr of the atom. Hash links are implemented by computing an address, called the hash-address, in a specified array, called the hash-array, and storing the hash-value and the hash-item into the cell with that address. The contents of that cell, i.e. the hash-value and hash-item, is then called the hash-link.[1]

Since the hash-array is obviously much smaller than the total number of possible hash-items,[2] the hash-address computed from item may already contain a hash-link. If this link is from item,[3] the new hash-value simply replaces the old hash-value. Otherwise, another hash-address (in the same hash-array) must be computed, etc, until an empty cell is found,[4] or a cell containing a hash-link from item.

When a hash link for item is being retrieved, the hash-address is computed

---

[1] The term hash link (unhyphenated) refers to the process of associating information this way, or the 'association' as an abstract concept.

[2] which is the total number of INTERLISP pointers, i.e., 256K.

[3] eq is used for comparing item with the hash-item in the cell.

[4] After a certain number of iterations (the exact algorithm is complicated), the hash-array is considered to be full, and the array is either enlarged, or an error is generated, as described below in the discussion of overflow.

using the same algorithm as that employed for making the hash link. If the corresponding cell is empty, there is no hash link for item. If it contains a hash-link from item, the hash-value is returned. Otherwise, another hash-address must be computed, and so forth.[5]

Note that more than one hash link can be associated with a given hash-item by using more than one hash-array.

Hash Link Functions

In the description of the functions below, the argument array has one of three forms: (1) NIL, in which case the hash-array provided by the system, syshasharray, is used;[6] (2) a hash-array created by the function harray, or created from an ordinary array using clrhash as described below; or (3) a list car of which is a hash-array. The latter form is used for specifying what is to be done on overflow, as described below.

harray[n] — creates a hash-array of size n, equivalent to clrhash[array[n]].

clrhash[array] — sets all elements of array to 0 and sets left half of first word of header to -1. Value is array.

puthash[item;val;array] — puts into array a hash-link from item to val.

---

[5] For reasonable operation, the hash array should be ten to twenty percent larger than the maximum number of hash links to be made to it.

[6] syshasharray is not used by the system, it is provided solely for the user's benefit. It is initially 512 words large, and is automatically enlarged by 50% whenever it is 'full'. See page 7.7.

Replaces previous link from same item, if any. If val=NIL any old link is removed, (hence a hash-value of NIL is not allowed). Value is val.

gethash[item;array] — finds hash-link from item in array, and returns the hash-value. Value is NIL if no link exists. gethash compiles open.

rehash[oldar;newar] — hashes all items and values in oldar into newar. The two arrays do not have to be (and usually aren't) the same size. Value is newar.

maphash[array;maphfn] — maphfn is a function of two arguments. For each hash-link in array, maphfn will be applied to the hash-value and hash-item, e.g. maphash[a;(LAMBDA(X Y) (AND(LISTP Y) (PRINT X)))] will print the hash-value for all hash-links from lists. The value of maphash is array.

dmphash[arrayname] — Nlambda-nospread that prints on the primary output file a loadable form which will restore what is in the hash-array specified by arrayname, e.g. (E (DMPHASH SYSHASHARRAY)) as a prettydef command will dump the system hash-array.

Note: all eq identities except atoms and small integers are lost by dumping and loading because read will create new structure for each item. Thus if two lists contain an eq substructure, when they are dumped and loaded back in, the corresponding substructures while equal are no longer eq.[7]

---

[7] circlprint and circlmaker (Section 21) provide a way of dumping and reloading structures containing eq substructures so that these identities are preserved.

Hash Overflow

By using an array argument of a special form, the user can provide for automatic enlargement of a hash-array when it overflows, i.e., is full and an attempt is made to store a hash link into it. The array argument is either of the form (hash-array . n), n a positive integer; or (hash-array . f), f a floating point number; or (hash-array). In the first case, a new hash-array is created with n more cells than the current hash-array. In the second case, the new hash array will be f times the size of the current hash-array. The third case, (hash-array), is equivalent to (hash-array . 1.5). In each case, the old hash-array is rplacaed into the dotted pair, and the computation continues.

If a hash-array overflows, and the array argument used was not one of these three forms, the error HASH TABLE FULL is generated, which will either cause a break or unwind to the last errorset, as per treatment of errors described in Section 16.

The system hash array, syshasharray, is automatically enlarged by 1.5 when it is full.

Index for Section 7

SECTION 8

FUNCTION DEFINITION AND EVALUATION

General Comments

A function definition in INTERLISP is stored in a special cell called the function definition cell, which is associated with each literal atom. This cell is directly accessible via the two functions putd, which puts a definition in the cell, and getd which gets the definition from the cell. In addition, the function fntyp returns the function type, i.e., EXPR, EXPR* ... FSUBR* as described in Section 4. Exprp, ccodep, and subrp are true if the function is an expr, compiled function, or subr respectively; argtype returns 0, 1, 2, or 3, depending on whether the function is a spread or nospread (i.e., its fntyp ends in *), or evaluate or no-evaluate (i.e., its fntyp begins with F or CF); arglist returns the list of arguments; and nargs returns the number of arguments. fntyp, exprp, ccodep, subrp, argtype, arglist, and nargs can be given either a literal atom, in which case they obtain the function definition from the atom's definition cell, or a function definition itself.

Subrs

Because subrs,[1] are called in a special way, their definitions are stored

---

[1] Basic functions, handcoded in machine language, e.g. cons, car, cond. The terms subr includes spread/nospread, eval/noeval functions, i.e. the four fntyp's SUBR, FSUBR, SUBR*, and FSUBR*.

differently than those of compiled or interpreted functions. In the right half of the definition cell is the address of the first instruction of the subr, and in the left half its argtype: 0, 1, 2, or 3. getd of a subr returns a dotted pair of argtype and address. Note that this is not the same word as appears in the definition cell, but a new cons; i.e., each getd of a subr performs a cons. Similarly, putd of a definition of the form (number . address), where number = 0, 1, 2, or 3, and address is in the appropriate range, stores the definition as a subr, i.e., takes the cons apart and stores car in the left half of the definition cell and cdr in the right half.

Validity of Definitions

Although the function definition cell is intended for function definitions, putd and getd do not make thorough checks on the validity of definitions that "look like" exprs, compiled code, or subrs. Thus if putd is given an array pointer, it treats it as compiled code, and simply stores the array pointer in the definition cell. getd will then return the array pointer. Similarly, a call to that function will simply transfer to what would normally be the entry point for the function, and produce random results if the array were not compiled function.

Similarly, if putd is given a dotted pair of the form (number . address) where number is 0, 1, 2, or 3, and address falls in the subr range, putd assumes it is a subr and stores it away as described earlier. getd would then return cons of the left and right half, i.e., a dotted pair equal (but not eq) to the expression originally given putd. Similarly, a call to this function would transfer to the corresponding address.

Finally, if putd is given any other list, it simply stores it away. A call to this function would then go through the interpreter as described in the appendix.

Note that putd does not actually check to see if the s-expression is valid definition, i.e., begins with LAMBDA or NLAMBDA. Similarly, exprp is true if a definition is a list and not of the form (number . address), number = 0, 1, 2, or 3 and address a subr address; subrp is true if it is of this form. arglist and nargs work correspondingly.

Only fntyp and argtype check function definitions further than that described above: both argtype and fntyp return NIL when exprp is true but car of the definition is not LAMBDA or NLAMBDA.[2] In other words, if the user uses putd to put (A B C) in a function definition cell, getd will return this value, the editor and prettyprint will both treat it as a definition, exprp will return T, ccodep and subrp NIL, arglist B, and nargs 1.

getd[x] — gets the function definition of x. Value is the definition.[3] Value is NIL if x is not a literal atom, or has no definition.

fgetd[x] — fast version of getd that compiles open as car[vag[add1[loc[x]]]]. Interpreted, generates an error, BAD ARGUMENT - FGETD, if x is not a literal atom.[4]

---

[2] These functions have different value on LAMBDAs and NLAMBDAs and hence must check. The compiler and interpreter also take different actions for LAMBDAs and NLAMBDAs, and therefore generate errors if the definition is neither.

[3] Note that getd of a subr performs a cons, as described on page 8.2. See footnote on fgetd below.

[4] Fgetd is intended primarily to check whether a function *has* a definition, rather than to obtain the definition. Therefore, for subrs, fgetd returns just the address of the function definition, not the dotted pair returned by getd, page 8.2, thereby saving the cons.

putd[x;y] — puts the definition y into x's function cell. Value is y. Generates an error, ILLEGAL ARG - PUTD, if x is not a literal atom, or y is a string, number, or literal atom other than NIL.

putdq[x;y] — nlambda version of putd; both arguments are considered quoted. Value is x.

movd[from;to;copyflg] — Moves the definition of from to to, i.e., redefines to. If copyflg=T, a copy of the definition of from is used. copyflg=T is only meaningful for exprs, although movd works for compiled functions and subrs as well. The value of movd is to.

Note: *fntyp, subrp, ccodep, exprp, argtype, nargs, and arglist all can be given either the name of a function, or a definition.*

fntyp[fn] — Value is NIL if fn is not a function definition or the name of a defined function. Otherwise fntyp returns one of the following as defined in the section on function types:

EXPR	CEXPR	SUBR
FEXPR	CFEXPR	FSUBR
EXPR*	CEXPR*	SUBR*
FEXPR*	CFEXPR*	FSUBR*

The prefix F indicates unevaluated arguments, the prefix C indicates compiled code; and the suffix * indicates an indefinite number of arguments.

fntyp returns FUNARG if fn is a funarg expression. See Section 11.

subrp[fn]

is true if and only if fntyp[fn] is either SUBR, FSUBR, SUBR*, or FSUBR*, i.e., the third column of fntyp's.

ccodep[fn]

is true if and only if fntyp[fn] is either CEXPR, CFEXPR, CEXPR*, or CFEXPR*, i.e., second column of fntyp's.

exprp[fn]

is true if fntyp[fn] is either EXPR, FEXPR, EXPR*, or FEXPR*, i.e., first column of fntyp's. However, exprp[fn] is also true if fn is (has) a list definition that is not a SUBR, but does not begin with either LAMBDA or NLAMBDA. In other words, exprp is not quite as selective as fntyp.

argtype[fn]

fn is the name of a function or its definition. The value of argtype is the argtype of fn, i.e., 0, 1, 2, or 3, or NIL if fn is not a function. The interpretation of the argtype is:

0 eval/spread function
(EXPR, CEXPR, SUBR)

1 no-eval/spread functions
(FEXPR, CFEXPR, FSUBR)

2 eval/nospread functions
(EXPR*, CEXPR*, SUBR*)

3 no-eval/nospread functions
(FEXPR*, CFEXPR*, FSUBR*)

i.e., argtype corresponds to the *rows* of fntyps.

nargs[fn] value is the number of arguments of fn, or NIL if fn is not a function.[5] nargs uses exprp, not fntyp, so that nargs[(A (B C) D)]=2. Note that if fn is a SUBR or FSUBR, nargs = 3, regardless of the number of arguments logically needed/used by the routine. If fn is a nospread function, nargs = 1.

arglist[fn] value is the 'argument list' for fn. Note that the 'argument list' is an atom for nospread functions. Since NIL is a possible value for arglist, an error is generated, ARGS NOT AVAILABLE, if fn is not a function.[6]

If fn is a SUBR or FSUBR, the value of arglist is (U V W), if a SUBR* or FSUBR*, the value is U. This is merely a 'feature' of arglist, subrs do not actually store the names U, V, or W on the stack. However, if the user breaks or traces a SUBR (Section 15), these will be the argument names used when an equivalent EXPR definition is constructed.

define[x] The argument of define is a list. Each element of the list is itself a list either of the form (name definition) or (name arguments ...). In the second case, following 'arguments' is the body of the definition. As an example, consider the

---

[5] i.e., if exprp, ccodep, and subrp are all NIL.

[6] If fn is a compiled function, the argument list is constructed, i.e. each call to arglist requires making a new list. For interpreted functions, the argument list is simply cadr of getd.

following two equivalent expressions for defining the function null.

1) (NULL (LAMBDA (X) (EQ X NIL)))

2) (NULL (X) (EQ X NIL))

define will generate an error on encountering an atom where a defining list is expected. If dfnflg=NIL, an attempt to *redefine* a function fn will cause define to print the message (fn REDEFINED) and to save the old definition of fn using savedef before redefining it. If dfnflg=T, the function is simply redefined. If dfnflg=PROP or ALLPROP, the new definition is stored on the property list under the property EXPR. (ALLPROP affects the operation of rpaqq and rpaq, section 5). dfnflg is initially NIL.

dfnflg is reset by load to enable various ways of handling the defining of functions and setting of variables when loading a file. For most applications, the user will not reset dfnflg directly himself.

Note: define will operate correctly if the function is already defined and broken, advised, or broken-in.

defineq[x_1;x_i;...;x_n] — nlambda nospread version of define, i.e., takes an indefinite number of arguments which are not evaluated. Each x_i must be a list, of the form described in define. defineq calls define, so dfnflg affects its operation the same as define.

savedef[fn] — Saves the definition of fn on its property list under property EXPR, CODE, or SUBR depending on its fntyp. Value is the property name used. If getd[fn] is non-NIL, but fntyp[fn] is NIL, saves on property name LIST. This situation can arise

when a function is redefined which was originally defined with LAMBDA misspelled or omitted.

If fn is a list, savedef operates on each function in the list, and its value is a list of the individual values.

unsavedef[fn;prop]

Restores the definition of fn from its property list under property prop (see savedef above). Value is prop. If nothing saved under prop, and fn is defined, returns (prop NOT FOUND), otherwise generates an error, NOT A FUNCTION.

If prop is not given, i.e. NIL, unsavedef looks under EXPR, CODE, and SUBR, in that order. The value of unsavedef is the property name, or if nothing is found and fn is a function, the value is (NOTHING FOUND); otherwise generates an error, NOT A FUNCTION.

If dfnflg=NIL, the current definition of fn, if any, is saved using savedef. Thus one can use unsavedef to switch back and forth between two definitions of the same function, keeping one on its property list and the other in the function definition cell.

If fn is a list, unsavedef operates on each function of the list, and its value is a list of the individual values.

eval[x][7]

eval evaluates the expression x and returns this value i.e. eval provides a way of calling the interpreter. Note that eval is itself a lambda type function, so *its* argument is the first evaluated, e.g.,

```
←SET(FOO (ADD1 3))
(ADD1 3)
←(EVAL FOO)
4
←EVAL(FOO)  or  (EVAL (QUOTE FOO))
(ADD1 3)
```

e[x]

nlambda nospread version of eval. Thus it eliminates the extra pair of parentheses for the list of arguments for eval. i.e., e x is equivalent to eval[x]. Note however that in INTERLISP, the user can type just x to get x evaluated. (See Section 3.)

apply[fn;args]

apply applies the function fn to the arguments args. The individual elements of args are not evaluated by apply, fn is simply called with args as its argument list.[8] Thus for the purposes of apply, nlambda's and lambda's are treated the same. However like eval, apply is a lambda function so *its* arguments are evaluated before it is called e.g.,

---

[7] eval is a subr so that the 'name' x does not actually appear on the stack.

[8] Note that fn may still explicitly evaluate one or more of its arguments itself, as in the case of setq. Thus (APPLY (QUOTE SETQ) (QUOTE (FOO (ADD1 3)))) will set FOO to 4, whereas (APPLY (QUOTE SET) (QUOTE (FOO (ADD1 3)))) will set FOO to the expression (ADD1 3).

```
←SET(FOO1 3)
3
←SET(FOO2 4)
4
←(APPLY (QUOTE IPLUS) (LIST FOO1 FOO2]
7
```

Here, foo1 and foo2 were evaluated when the second argument to apply was evaluated. Compare with:

```
←SET(FOO1 (ADD1 2))
(ADD1 2)
←SET(FOO2 (SUB1 5))
(SUB1 5)
←(APPLY (QUOTE IPLUS) (LIST FOO1 FOO2]

NON-NUMERIC ARG
(ADD1 2)
```

apply*[fn;arg_1;...;arg_n]

equivalent to apply[fn;list[arg_1;...;arg_n]] For example, if fn is the name of a functional argument to be applied to x and y, one can write (APPLY* FN X Y), which is equivalent to (APPLY FN (LIST X Y)). Note that (FN X Y) specifies a call to the function FN itself, and will cause an error if FN is not defined. (See Section 16.) FN will *not* be evaluated.

eval a[x;a]

Simulates a-list evaluation as in LISP 1.5. x is a form, a is a list of dotted pairs of variable name and value. a is 'spread' on the stack, and then x is evaluated, i.e., any variables appearing free in x, that also appears as car of an element of a will be given the value in the cdr of that element.

rpt[rptn;rptf]

Evaluates the expression rptf rptn times. At any point, rptn is the number of evaluations yet to

take place. Returns the value of the last evaluation. If rptn $\leq$ 0, rptf is not evaluated, and the value of rpt is NIL.

Note: rpt is a lambda function, so both its arguments are evaluated before rpt is called. For most applications, the user will probably want to use rptq.

rptq[rptn;rptf]

nlambda version of rpt: rptn is evaluated, rptf is not, e.g. (RPTQ 10 (READ)) will perform ten calls to read. rptq compiles open.

arg[var;m]

Used to access the individual arguments of a lambda nospread function. arg is an nlambda function used like set var is the *name* of the atomic argument list to a lambda-nospread function, and is not evaluated; m is the number of the desired argument, and is evaluated. For example, consider the following definition of iplus in terms of plus.

```
[LAMBDA X
  (PROG ((M 0)
         (N 0))
    LP  (COND
          ((EQ N X)
            (RETURN M)))
        (SETQ N (ADD1 N))
        [SETQ M (PLUS M (ARG X N)))
        (GO LP]
```

The value of arg is undefined for m less than or equal to 0 or greater than the *value* of var.[9] Lower numbered arguments appear earlier in the form, e.g. for (IPLUS A B C),

---

[9] For lambda nospread functions, the lambda variable is bound to the number of arguments actually given to the function. See Section 4.

arg[X;1]=the value of A,

arg[X;2]=the value of B, and

arg[X;3]=the value of C.

Note that the lambda variable should *never* be reset. However, individual arguments can be reset using setarg described below.

setarg[var;m;x]

sets to x the mth argument for the lambda nospread function whose argument list is var. var is considered quoted, m and x are evaluated; e.g. in the previous example, (SETARG X (ADD1 N)(MINUS M)) would be an example of the correct form for setarg.

Index for Section 8

SECTION 9

THE INTERLISP EDITOR[1]

The INTERLISP editor allows rapid, convenient modification of list structures. Most often it is used to edit function definitions, (often while the function itself is running) via the function editf, e.g., EDITF(FOO). However, the editor can also be used to edit the value of a variable, via editv, to edit a property list, via editp, or to edit an arbitrary expression, via edite. It is an important feature which allows good on-line interaction in the INTERLISP system.

This chapter begins with a lengthy introduction intended for the new user. The reference portion begins on page 9.15.

9.1 Introduction

Let us introduce some of the basic editor commands, and give a flavor for the editor's language structure by guiding the reader through a hypothetical editing session. Suppose we are editing the following incorrect definition of append:

---

[1] The editor was written by and is the responsibility of W. Teitelman.

```
[LAMBDA (X)
  Y
  (COND
    ((NUL X)
      Z)
    (T (CONS (CAR)
             (APPEND (CDR X Y]
```

We call the editor via the function editf:

```
←EDITF(APPEND)
EDIT
*
```

The editor responds by typing EDIT followed by *, which is the editor's prompt character, i.e., it signifies that the editor is ready to accept commands.[2]

At any given moment, the editor's attention is centered on some substructure of the expression being edited. This substructure is called the *current expression*, and it is what the user sees when he gives the editor the command P, for print. Initially, the current expression is the top level one, i.e., the entire expression being edited. Thus:

```
*P
(LAMBDA (X) Y (COND & &))
*
```

Note that the editor prints the current expression as though printlevel were set to 2, i.e., sublists of sublists are printed as &. The command ? will print the current expression as though printlevel were 1000.

```
*?
(LAMBDA (X) Y (COND ((NUL X) Z) (T (CONS (CAR) (APPEND (CDR X Y))))))
*
```

and the command PP will prettyprint the current expression.

---

[2] In other words, all lines beginning with * were typed by the user, the rest by the editor.

A positive integer is interpreted by the editor as a command to descend into the correspondingly numbered element of the current expression. Thus:

```
*2
*p
(X)
*
```

A negative integer has a similar effect, but counting begins from the end of the current expression and proceeds backward, i.e., -1 refers to the last element in the current expression, -2 the next to the last, etc. For either positive integer or negative integer, if there is no such element, an error occurs,[3] the editor types the faulty command followed by a ?, and then another *. *The current expression is never changed when a command causes an error.* Thus:

```
*p
(X)
*2

2  ?
*1
*p
X
*
```

A phrase of the form 'the current expression is changed' or 'the current expression becomes' refers to a shift in the editor's attention, not to a modification of the structure being edited.

When the user changes the current expression by descending into it, the old current expression is not lost. Instead, the editor actually operates by

---

[3] 'Editor errors' are not of the flavor described in Section 16, i.e., they never cause breaks or even go through the error machinery but are direct calls to error! indicating that a command is in some way faulty. What happens next depends on the context in which the command was being executed. For example, there are conditional commands which branch on errors. In most situations, though, an error will cause the editor to type the faulty command followed by a ? and wait for more input. Note that typing control-E while a command is being executed aborts the command exactly as though it had caused an error.

maintaining a *chain* of expressions leading to the current one. The current expression is simply the last link in the chain. Descending adds the indicated subexpression onto the end of the chain, thereby making it be the current expression. The command 0 is used to ascend the chain; it removes the last link of the chain, thereby making the *previous* link be the current expression. Thus:

```
*P
X
*0 P
(X)
*0 -1 P
(COND (& Z) (T &))
*
```

Note the use of several commands on a single line in the previous output. The editor operates in a line buffered mode, the same as evalqt. Thus no command is actually seen by the editor, or executed, until the line is terminated, either by a carriage return, or a matching right parenthesis. The user can thus use control-A and control-Q for line-editing edit commands, the same as he does for inputs to evalqt.

In our editing session, we will make the following corrections to append: delete Y from where it appears, add Y to the end of the argument list,[4] change NUL to NULL, change Z to Y, add Z after CAR, and insert a right parenthesis following CDR X.

First we will delete Y. By now we have forgotten where we are in the function definition, but we want to be at the "top" so we use the command ↑, which ascends through the entire chain of expressions to the top level expression,

---

[4] These two operations could be though of as one operation, i.e., MOVE Y from its current position to a new position, and in fact there is a MOVE command in the editor. However, for the purposes of this introduction, we will confine ourselves to the simpler edit commands.

which then becomes the current expression, i.e., ↑ removes all links except the first one.

```
*↑ P
(LAMBDA (X) Y (COND & &))
*
```

Note that if we are already at the top, ↑ has no effect, i.e., it is a NOP. However, 0 would generate an error. In other words, ↑ means "go to the top," while 0 means "ascend one link."

The basic structure modification commands in the editor are:

(n) — $n \geq 1$ deletes the corresponding element from the current expression.

($n\ e_1 \ldots e_m$) — $n,m \geq 1$ replaces the <u>n</u>th element in the current expression with $e_1 \ldots e_m$.

($-n\ e_1 \ldots e_m$) — $n,m \geq 1$ inserts $e_1 \ldots e_m$ before the <u>n</u>th element in the current expression.

Thus:

```
*P
(LAMBDA (X) Y (COND & &))
*(3)
*(2 (X Y))
*P
(LAMBDA (X Y) (COND & &))
*
```

All structure modification done by the editor is destructive, i.e., the editor uses ***rplaca*** *and* ***rplacd*** *to physically change the structure it was given.*

Note that all three of the above commands perform their operation with respect

to the nth element from the front of the current expression; the sign of n is used to specify whether the operation is replacement or insertion. Thus, there is no way to specify deletion or replacement of the nth element from the end of the current expression, or insertion before the nth element from the end without counting out that element's position from the front of the list. Similarly, because we cannot specify insertion after a particular element, we cannot attach something at the end of the current expression using the above commands. Instead, we use the command N (for nconc). Thus we could have performed the above changes instead by:

```
*p
(LAMBDA (X) Y (COND & &))
*(3)
*2 (N Y)
*p
(X Y)
*↑ p
*(LAMBDA (X Y) (COND & &))
*
```

Now we are ready to change NUL to NULL. Rather than specify the sequence of descent commands necessary to reach NUL, and then replace it with NULL, e.g., 3 2 1 (1 NULL), we will use F, the find command, to find NUL:

```
*p
(LAMBDA (X Y) (COND & &))
*F NUL
*p
(NUL X)
*(1 NULL)
*0 P
((NULL X) Z)
*
```

Note that F is special in that it corresponds to *two* inputs. In other words, F says to the editor, "treat your *next* command as an expression to be searched for." The search is carried out in printout order in the current expression. If the target expression is not found there, F automatically ascends and searches those portions of the higher expressions that would appear after (in a printout) the current expression. If the search is successful, the new current

expression will be the structure where the expression was found,[5] and the chain will be the same as one resulting from the appropriate sequence of ascent and descent commands. If the search is not successful, an error occurs, and neither the current expression nor the chain is changed:[6]

```
*P
((NULL X) Z)
*F COND P

COND ?
*P
*((NULL X) Z)
*
```

Here the search failed to find a cond following the current expression, although of course a cond does appear earlier in the structure. This last example illustrates another facet of the error recovery mechanism: to avoid further confusion when an error occurs, all commands on the line *beyond* the one which caused the error (and all commands that may have been typed ahead while the editor was computing) are forgotten.[7]

We could also have used the R command (for replace) to change NUL to NULL. A command of the form (R e_1 e_2) will replace all occurrences of e_1 in the current expression by e_2. There must be at least one such occurrence or the R command will generate an error. Let us use the R command to change all Z's (even though there is only one) in append to Y:

---

[5] If the search is for an atom, e.g., F NUL, the current expression will be the structure containing the atom.

[6] F is never a NOP, i.e., if successful, the current expression after the search will never be the same as the current expression before the search. Thus F expr repeated without intervening commands that change the edit chain can be used to find successive instances of expr.

[7] i.e. the input buffer is cleared (and saved) (see clearbuf, Section 14). It can be restored, and the type-ahead recovered via the command $BUFS (alt-mode BUFS), described in Section 22.

```
*↑ (R Z Y)
*F Z

Z ?
*PP
  [LAMBDA (X Y)
    (COND
      ((NULL X)
        Y)
      (T (CONS (CAR)
                 (APPEND (CDR X Y]
*
```

The next task is to change (CAR) to (CAR X). We could do this by (R (CAR) (CAR X)), or by:

```
*F CAR
*(N X)
*P
(CAR X)
*
```

The expression we now want to change is the next expression after the current expression, i.e., we are currently looking at (CAR X) in (CONS (CAR X) (APPEND (CDR X Y))). We could get to the <u>append</u> expression by typing 0 and then 3 or -1, or we can use the command NX, which does both operations:

```
*P
(CAR X)
*NX P
(APPEND (CDR X Y))
*
```

Finally, to change (APPEND (CDR X Y)) to (APPEND (CDR X) Y), we could perform (2 (CDR X) Y), or (2 (CDR X)) and (N Y), or 2 and (3), deleting the Y, and then 0 (N Y). However, if Y were a complex expression, we would not want to have to retype it. Instead, we could use a command which effectively inserts and/or removes left and right parentheses. There are six of these commands: BI,BO,LI,LO,RI, and RO, for <u>b</u>oth <u>i</u>n, <u>b</u>oth <u>o</u>ut, <u>l</u>eft <u>i</u>n, <u>l</u>eft <u>o</u>ut, <u>r</u>ight <u>i</u>n, and <u>r</u>ight <u>o</u>ut. Of course, we will always have the same number of left parentheses as right parentheses, because the parentheses are just a notational guide to

structure that is provided by our print program.[8] Thus, left in, left out, right in, and right out actually do not insert or remove just one parenthesis, but this is very suggestive of what actually happens.

In this case, we would like a right parenthesis to appear following X in (CDR X Y). Therefore, we use the command (RI 2 2), which means insert a right parentheses after the second element in the second element (of the current expression):

```
*p
(APPEND (CDR X Y))
*(RI 2 2)
*p
(APPEND (CDR X) Y)
*
```

We have now finished our editing, and can exit from the editor, to test append, or we could test it while still inside of the editor, by using the E command:

```
*E APPEND((A B) (C D E))
(A B C D E)
*
```

The E command causes the next input to be given to evalqt. If there is another input following it, as in the above example, the first will be applied (apply) to the second. Otherwise, the input is evaluated (eval).

We prettyprint append, and leave the editor.

---

[8] Herein lies one of the principal advantages of a LISP oriented editor over a text editor: unbalanced parentheses errors are not possible.

```
*PP
  [LAMBDA (X Y)
    (COND
      ((NULL X)
        Y)
      (T (CONS (CAR X)
               (APPEND (CDR X) Y]
*OK
APPEND
←
```

9.2 Commands for the New User

As mentioned earlier, the INTERLISP manual is intended primarily as a reference manual, and the remainder of this chapter is organized and presented accordingly. While the commands introduced in the previous scenario constitute a complete set, i.e., the user could perform any and all editing operations using just those commands, there are many situations in which knowing the right command(s) can save the user considerable effort. We include here as part of the introduction a list of those commands which are not only frequently applicable but also easy to use. They are not presented in any particular order, and are all discussed in detail in the reference portion of the chapter.

UNDO — undoes the last modification to the structure being edited, e.g., if the user deletes the wrong element, UNDO will restore it. The availability of UNDO should give the user confidence to experiment with any and all editing commands, no matter how complex, because he can always reverse the effect of the command.

BK — like NX, except makes the expression immediately *before* the current expression become current.

BF — backwards find. Like F, except searches backwards, i.e., in inverse print order.

\   Restores the current expression to the expression before the last "big jump", e.g., a find command, an ↑, or another \. For example, if the user types F COND, and then F CAR, \ would take him back to the COND. Another \ would take him back to the CAR.

\P   like \ except it restores the edit chain to its state as of the last print, either by P, ?, or PP. If the edit chain has not been changed since the last print, \P restores it to its state as of the printing before that one, i.e., two chains are always saved.

Thus if the user types P followed by 3 2 1 P, \P will take him back to the first P, i.e., would be equivalent to 0 0 0. Another \P would then take him back to the second P. Thus the user can use \P to flip back and forth between two current expressions.

&,--   The search expression given to the F or BF command need not be a literal S-expression. Instead, it can be a pattern. The symbol & can be used anywhere within this pattern to match with any single *element* of a list, and -- can be used to match with any *segment* of a list. Thus, in the incorrect definition of append used earlier, F (NUL &) could have been used to find (NUL X), and F (CDR --) or F (CDR & &), but not F (CDR &), to find (CDR X Y).

Note that & and -- can be nested arbitrarily deeply in the pattern. For

example, if there are many places where the variable X is set, F SETQ may not find the desired expression, nor may F (SETQ X &). It may be necessary to use F (SETQ X (LIST --)). However, the usual technique in such a case is to pick out a unique atom which occurs prior to the desired expression, and perform two F commands. This "homing in" process seems to be more convenient than ultra-precise specification of the pattern.

$ (alt-mode)

$ is equivalent to -- at the character level, e.g. VER$ will match with VERYLONGATOM, as will $ATOM, $LONG$, (but not $LONG) and VNM. $ can be nested inside of a pattern, e.g., F (SETQ VER$ (CONS --)).

If the search is successful, the editor will print = followed by the atom which matched with the $-atom, e.g.,

```
*F (SETQ VER$ &)
=VERYLONGATOM
*
```

Frequently the user will want to replace the entire current expression, or insert something before it. In order to do this using a command of the form (n e_1 ... e_m) or (-n e_1 ... e_m), the user must be *above* the current expression. In other words, he would have to perform a 0 followed by a command with the appropriate number. However, if he has reached the current expression via an F command, he may not know what that number is. In this case, the user would like a command whose effect would be to modify the edit chain so that the current expression became the first element in a new, higher current expression. Then he could perform the desired operation via (1 e_1 ... e_m) or (-1 e_1 ... e_m). UP is provided for this purpose.

UP after UP operates, the old current expression is the first element of the new current expression. Note that if the current expression happens to be the first element in the next higher expression, then UP is exactly the same as 0. Otherwise, UP modifies the edit chain so that the new current expression is a tail[9] of the next higher expression:

```
*F APPEND P
(APPEND (CDR X) Y)
*UP P
... (APPEND & Y))
*0 P
(CONS (CAR X) (APPEND & Y))
*
```

The ... is used by the editor to indicate that the current expression is a *tail* of the next higher expression as opposed to being an element (i.e., a member) of the next higher expression. Note: if the current expression is *already* a tail, UP has no effect.

(B e_1 ... e_m) inserts e_1 ... e_m before the current expression, i.e., does an UP and then a -1.

(A e_1 ... e_m) inserts e_1 ... e_m after the current expression, i.e., does an UP and then either a (-2 e_1 ... e_m) or an (N e_1 ... e_m), if the current expression is the last one in the next higher expression.

---

9 Throughout this chapter 'tail' means 'proper tail' (see Section 5).

(: e_1 ... e_m) replaces current expression by e_1 ... e_m, i.e., does an UP and then a (1 e_1 ... e_m).

DELETE deletes current expression; equivalent to (:).

Earlier, we introduced the RI command in the append example. The rest of the commands in this family: BI, BO, LI, LO, and RO, perform similar functions and are useful in certain situations. In addition, the commands MBD and XTR can be used to combine the effects of several commands of the BI-BO family. MBD is used to embed the current expression in a larger expression. For example, if the current expression is (PRINT bigexpression), and the user wants to replace it by (COND (FLG (PRINT bigexpression))), he could accomplish this by (LI 1), (-1 FLG), (LI 1), and (-1 COND), or by a single MBD command, page 9.47.

XTR is used to extract an expression from the current expression. For example, extracting the PRINT expression from the above COND could be accomplished by (1), (LO 1), (1), and (LO 1) or by a single XTR command. The new user is encouraged to include XTR and MBD in his repertoire as soon as he is familiar with the more basic commands.

This ends the introductory material.

9.3 Attention Changing Commands

Commands to the editor fall into three classes: commands that change the current expression (i.e., change the edit chain) thereby "shifting the editor's attention," commands that modify the structure being edited, and miscellaneous commands, e.g., exiting from the editor, printing, evaluating expressions, etc.

Within the context of commands that shift the editor's attention, we can distinguish among (1) those commands whose operation depends only on the *structure* of the edit chain, e.g., 0, UP, NX; (2) those which depend on the *contents* of the structure, i.e., commands that search; and (3) those commands which simply restore the edit chain to some previous state, e.g., \, \P. (1) and (2) can also be thought of as local, small steps versus open ended, big jumps. Commands of type (1) are discussed on page 9.15-21, type (2) on page 9.21-34, and type (3) on page 9.34-36.

9.3.1 Local Attention-Changing Commands

UP — (1) If a P command would cause the editor to type ... before typing the current expression, i.e. the current expression is a tail of the next higher expression, UP has no effect; otherwise

(2) UP modifies the edit chain so that the old current expression (i.e., the one at the time UP was called) is the first element in the new current expression.[10]

---

[10] If the current expression is the first element in the next higher expression UP simply does a 0. Otherwise UP adds the corresponding tail to the edit chain.

Examples: The current expression in each case is

(COND ((NULL X) (RETURN Y))).

1.
```
*1 P
COND
*UP P
(COND (& &))
```

2.
```
*-1 P
((NULL X) (RETURN Y))
*UP P
... ((NULL X) (RETURN Y))
*UP P
... ((NULL X) (RETURN Y)))
```

3.
```
*F NULL P
(NULL X)
*UP P
((NULL X) (RETURN Y))
*UP P
... ((NULL X) (RETURN Y)))
```

The execution of UP is straightforward, except in those cases where the current expression appears more than once in the next higher expression. For example, if the current expression is (A NIL B NIL C NIL) and the user performs 4 followed by UP, the current expression should then be ... NIL C NIL). UP can determine which tail is the correct one because the commands that descend save the last tail on an internal editor variable, lastail. Thus after the 4 command is executed, lastail is (NIL C NIL). When UP is called, it first determines if the current expression is a tail of the next higher expression. If it is, UP is finished. Otherwise, UP computes memb[current-expression;next-higher-expression] to obtain a tail beginning with the current expression.[11] If there are no other instances of the current expression in the next higher expression, this tail is the correct one.

---

[11] The current expression should *always* be either a tail or an element of the next higher expression. If it is neither, for example the user has directly (and incorrectly) manipulated the edit chain, UP generates an error.

Otherwise UP uses lastail to select the correct tail.[12]

n (n $\geq$ 1) — adds the nth element of the current expression to the front of the edit chain, thereby making it be the new current expression. Sets lastail for use by UP. Generates an error if the current expression is not a list that contains at least n elements.

-n (n $\geq$ 1) — adds the nth element from the end of the current expression to the front of the edit chain, thereby making it be the new current expression. Sets lastail for use by UP. Generates an error if the current expression is not a list that contains at least n elements.

0 — Sets edit chain to cdr of edit chain, thereby making the next higher expression be the new current expression. Generates an error if there is no higher expression, i.e. cdr of edit chain is NIL.

Note that 0 usually corresponds to going back to the next higher left

---

[12] Occasionally the user can get the edit chain into a state where lastail cannot resolve the ambiguity, for example if there were two non-atomic structures in the same expression that were eq, and the user descended more than one level into one of them and then tried to come back out using UP. In this case, UP prints LOCATION UNCERTAIN and generates an error. Of course, we could have solved this problem completely in our implementation by saving at each descent *both* elements and tails. However, this would be a costly solution to a situation that arises infrequently, and when it does, has no detrimental effects. The lastail solution is cheap and resolves 99% of the ambiguities.

parenthesis, but not always. For example, if the current expression is (A B C D E F B), and the user performs:

```
*3 UP P
... C D E F G)
*3 UP P
... E F G)
*0 P
... C D E F G)
```

If the intention is to go back to the next higher left parenthesis, regardless of any intervening tails, the command !0 can be used.[13]

!0	does repeated 0's until it reaches a point where the current expression is *not* a tail of the next higher expression, i.e., always goes back to the next higher left parenthesis.
↑	sets edit chain to <u>last</u> of edit chain, thereby making the top level expression be the current expression. Never generates an error.
NX	effectively does an UP followed by a 2,[14] thereby making the current expression be the next expression. Generates an error if the current expression is the last one in a list. (However, !NX described below will handle this case.)
BK	makes the current expression be the previous

---

[13] !0 is pronounced bang-zero.

[14] Both NX and BK operate by performing a !0 followed by an appropriate number, i.e. there won't be an extra tail above the new current expression, as there would be if NX operated by performing an UP followed by a 2.

expression in the next higher expression. Generates an error if the current expression is the first expression in a list.

For example, if the current expression is (COND ((NULL X) (RETURN Y))):

```
*F RETURN P
(RETURN Y)
*BK P
(NULL X)
```

(NX n) n ≥ 1 — equivalent to n NX commands, except if an error occurs, the edit chain is not changed.

(BK n) n ≥ 1 — equivalent to n BK commands, except if an error occurs, the edit chain is not changed.

Note: (NX -n) is equivalent to (BK n), and vice versa.

!NX — makes current expression be the next expression at a higher level, i.e., goes through any number of right parentheses to get to the next expression.

For example:

```
*PP
  (PROG ((L L)
         (UF L))
     LP  (COND
           ((NULL (SETQ L (CDR L)))
              (ERROR!))
           ([NULL (CDR (FMEMB (CAR L)
                              (CADR L]
             (GO LP)))
         (EDITCOM (QUOTE NX))
         (SETQ UNFIND UF)
         (RETURN L))
*F CDR P
(CDR L)
*NX

NX   ?
*!NX P
(ERROR!)
*!NX P
((NULL &) (GO LP))
*!NX P
(EDITCOM (QUOTE NX))
*
```

!NX operates by doing 0's until it reaches a stage where the current expression is *not* the last expression in the next higher expression, and then does a NX. Thus !NX always goes through at least one unmatched right parenthesis, and the new current expression is always on a different level, i.e., !NX and NX always produce different results. For example using the previous current expression:

```
*F CAR P
(CAR L)
*!NX P
(GO LP)
*\P P
(CAR L)
*NX P
(CADR L)
*
```

(NTH n) n ≠ 0 — equivalent to n followed by UP, i.e., causes the list starting with the nth element of the current expression (or nth from the end if n < 0) to

become the current expression.[15] Causes an error if current expression does not have at least n elements.

A generalized form of NTH using location specifications is described on page 9.32.

9.3.2 Commands That Search

All of the editor commands that search use the same pattern matching routine.[16] We will therefore begin our discussion of searching by describing the pattern match mechanism. A pattern pat matches with x if:

1. pat is eq to x.
2. pat is &.
3. pat is a number and eqp to x.
4. pat is a string and strequal[pat;x] is true.
5. If car[pat] is the atom *ANY*, cdr[pat] is a list of patterns and pat matches x if and only if one of the patterns on cdr[pat] matches x.

6a. If pat is a literal atom or string containing one or more alt-modes, each $ can match an indefinite number (including 0) of contiguous characters in a literal atom or string, e.g. VER$ matches both VERYLONGATOM and "VERYLONGSTRING" as do $LONG$ (but not $LONG), and VLT.

---

[15] (NTH 1) is a NOP, as is (NTH -n) where n is the length of the current expression.

[16] This routine is available to the user directly, and is described on page 9.88.

6b. If pat is a literal atom or string ending in *two* alt-modes, pat matches with the first atom or string that is "close" to pat, in the sense used by the spelling corrector (Section 17). E.g. CONSS$$ matches with CONS, CNONC$$ with NCONC or NCONC1.

The pattern matching routine always types a message of the form =x to inform the user of the object matched by a pattern of type 6a or 6b,[17] e.g. =VERYLONGATOM.

7. If car[pat] is the atom --, pat matches x if

   a. cdr[pat]=NIL, i.e. pat=(--), e.g.

      (A --) matches (A) (A B C) and (A . B)

      In other words, -- can match any tail of a list.

   b. cdr[pat] matches with some tail of x,

      e.g. (A -- (&)) will match with (A B C (D)),

      but not (A B C D), or (A B C (D) E). However,

      note that (A -- (&) --) will match with

      (A B C (D) E).

      In other words, -- can match any interior segment of a list.

8. If car[pat] is the atom ==, pat matches x if and only if cdr[pat] is eq to x.[18]

9. Otherwise if x is a list, pat matches x if car[pat] matches car[x], and cdr[pat] matches cdr[x].

When the editor is searching, the pattern matching routine is called to match with *elements* in the structure, unless the pattern begins with ..., in which case cdr of the pattern is matched against proper tails in the structure. Thus if the current expression is (A B C (B C)),

---

[17] unless editquietflg=T.

[18] Pattern 8 is for use by programs that call the editor as a subroutine, since any non-atomic expression in a command *typed* in by the user obviously cannot be eq to already existing structure.

```
*F (B --)
*P (B C)
*0 F (... B --)
*P
... B C (B C))
```

Matching is also attempted with atomic tails (except for NIL). Thus

```
*P
(A (B . C))
*F C
*P
... . C)
```

Although the current expression is the atom C after the final command, it is printed as ... . C) to alert the user to the fact that C is a *tail*, not an element. Note that the pattern C will match with either instance of C in (A C (B . C)), whereas (... . C) will match only the second C. The pattern NIL will only match with NIL as an element, i.e. it will not match in (A B), even though cddr of (A B) is NIL. However, (... . NIL) (or equivalently (...)) may be used to specify a NIL *tail*, e.g. (... . NIL) will match with cdr of the third subexpression of ((A . B) (C . D) (E)).

Search Algorithm

Searching begins with the current expression and proceeds in print order. Searching usually means find the next instance of this pattern, and consequently a match is not attempted that would leave the edit chain unchanged.[19] At each step, the pattern is matched against the next element in the expression currently being searched, unless the pattern begins with ... in which case it is matched against the next tail of the expression.

---

19 However, there is a version of the find command which can succeed and leave the current expression unchanged (see page 9.26).

If the match is not successful, the search operation is recursive first in the car direction and then in the cdr direction, i.e., if the element under examination is a list, the search descends into that list before attempting to match with other elements (or tails) at the same level.[20]

However, at no point is the total recursive depth of the search (sum of number of cars and cdrs descended into) allowed to exceed the value of the variable maxlevel. At that point, the search of that element or tail is abandoned, exactly as though the element or tail had been completely searched without finding a match, and the search continues with the element or tail for which the recursive depth is below maxlevel. This feature is designed to enable the user to search circular list structures (by setting maxlevel small), as well as protecting him from accidentally encountering a circular list structure in the course of normal editing. maxlevel is initially set to 300.[21]

If a successful match is not found in the current expression, the search automatically ascends to the next higher expression,[22] and continues searching there on the next expression after the expression it just finished searching. If there is none, it ascends again, etc. This process continues until the entire edit chain has been searched, at which point the search fails, and an error is generated. If the search fails (or, what is equivalent, is aborted by control-E), the edit chain is not changed (nor are any conses performed).

If the search is successful, i.e., an expression is found that the pattern

---

[20] There is also a version of the find command (see page 9.27) which only attempts matches at the top level of the current expression, i.e., does not descend into elements, or ascend to higher expressions.

[21] maxlevel can also be set to NIL, which is equivalent to infinity.

[22] See footnote on page 9.24.

matches, the edit chain is set to the value it would have had had the user reached that expression via a sequence of integer commands.

If the expression that matched was a list, it will be the final link in the edit chain, i.e., the new current expression. If the expression that matched is not a list, e.g., is an atom, the current expression will be the tail beginning with that atom,[23] i.e., that atom will be the first element in the new current expression. In other words, the search effectively does an *UP*.[24]

Search Commands

All of the commands below set lastail for use by UP, set unfind for use by \ (page 9.35), and do not change the edit chain or perform any conses if they are unsuccessful or aborted.

F pattern

i.e., two commands: the F informs the editor that the *next* command is to be interpreted as a pattern. This is the most common and useful form of the find command. If successful, the edit chain always changes, i.e., F pattern means find the next instance of pattern.

If memb[pattern;current-expression] is true, F does not proceed with a full recursive search. If the value of the memb is NIL, F invokes the search algorithm described earlier.

---

[23] Unless the atom is a tail, e.g. B in (A . B). In this case, the current expression will be B, but will print as ... . B).

[24] Unless upfindflg=NIL (initially set to T). For discussion, see page 9.43-44.

Thus if the current expression is (PROG NIL LP (COND (-- (GO LP1))) ... LP1 ...), F LP1 will find the prog label, not the LP1 inside of the GO expression, even though the latter appears first (in print order) in the current expression. Note that 1 (making the atom PROG be the current expression), followed by F LP1 *would* find the first LP1.

(F pattern N)

same as F pattern, i.e., finds the next instance of pattern, except the memb check of F pattern is not performed.

(F pattern T)

Similar to F pattern, except may succeed without changing edit chain, and does not perform the memb check.

Thus if the current expression is (COND ..), F COND will look for the next COND, but (F COND T) will 'stay here'.

(F pattern n) n ≥ 1

Finds the nth place that pattern matches. Equivalent to (F pattern T) followed by (F pattern N) repeated n-1 times. Each time pattern successfully matches, n is decremented by 1, and the search continues, until n reaches 0. Note that the pattern does not have to match with n identical expressions; it just has to match n times. Thus if the current expression is (FOO1 FOO2 FOO3), (F FOO$ 3) will find FOO3.

If the pattern does not match successfully n times, an error is generated and the edit chain is unchanged (even if the pattern matched n-1 times).

(F pattern) or
(F pattern NIL)

only matches with elements at the top level of the current expression, i.e., the search will not descend into the current expression, nor will it go outside of the current expression. May succeed without changing edit chain.

For example, if the current expression is (PROG NIL (SETQ X (COND & &)) (COND &) ...), F COND will find the COND inside the SETQ, whereas (F (COND --)) will find the top level COND, i.e., the second one.

(FS $pattern_1$ ... $pattern_n$)

equivalent to F $pattern_1$ followed by F $pattern_2$ ... followed by F $pattern_n$, so that if F $pattern_m$ fails, edit chain is left at place $pattern_{m-1}$ matched.

(F= expression x)

equivalent to (F (== . expression) x), i.e., searches for a structure eq to expression, see page 9.22.

(ORF $pattern_1$ ... $pattern_n$)

equivalent to (F (*ANY* $pattern_1$ ... $pattern_n$) N), i.e., searches for an expression that is matched by either $pattern_1$, $pattern_2$, ... or $pattern_n$. See page 9.21.

BF pattern

backwards find. Searches in reverse print order, beginning with expression immediately before the current expression (unless the current expression is the top level expression, in which case BF searches the entire expression, in reverse order).

BF uses the same pattern match routine as F, and maxlevel and upfindflg have the same effect, but the searching begins at the *end* of each list, and descends into each element before attempting to match that element. If unsuccessful, the search continues with the next previous element, etc., until the front of the list is reached, at which point BF ascends and backs up, etc.

For example, if the current expression is (PROG NIL (SETQ X (SETQ Y (LIST Z))) (COND ((SETQ W --) --)) --), F LIST followed by BF SETQ will leave the current expression as (SETQ Y (LIST Z)), as will F COND followed by BF SETQ.

(BF pattern T) — search always includes current expression, i.e., starts at the end of current expression and works backward, then ascends and backs up, etc.

Thus in the previous example, where F COND followeBF SETQ found (SETQ Y (LIST Z)), F COND followed by (BF SETQ T) would find the (SETQ W --) expression.

(BF pattern)
(BF pattern NIL) — same as BF pattern.

Location Specification

Many of the more sophisticated commands described later in this chapter use a more general method of specifying position called a location specification. A location specification is a list of edit commands that are executed in the normal fashion with two exceptions. First, all commands not recognized by the

editor are interpreted as though they had been preceded by F.[25] For example, the location specification (COND 2 3) specifies the 3rd element in the first clause of the next COND.[26]

Secondly, if an error occurs while evaluating one of the commands in the location specification, and the edit chain had been changed, i.e., was not the same as it was at the beginning of that execution of the location specification, the location operation will continue. In other words, the location operation keeps going unless it reaches a state where it detects that it is 'looping', at which point it gives up. Thus, if (COND 2 3) is being located, and the first clause of the next COND contained only two elements, the execution of the command 3 would cause an error. The search would then continue by looking for the next COND. However, if a point were reached where there were no further CONDs, then the first command, COND, would cause the error; the edit chain would not have been changed, and so the entire location operation would fail, and cause an error.

The IF command in conjunction with the ## function provide a way of using arbitrary predicates applied to elements in the current expression. IF and ## will be described in detail later in the chapter, along with examples illustrating their use in location specifications.

Throughout this chapter, the meta-symbol @ is used to denote a location specification. Thus @ is a list of commands interpreted as described above. @ can also be atomic, in which case it is interpreted as list[@].

---

[25] Normally such commands would cause errors.

[26] Note that the user could always write F COND followed by 2 and 3 for (COND 2 3) if he were not sure whether or not COND was the name of an atomic command.

(LC . @) provides a way of explicitly invoking the location operation, e.g. (LC COND 2 3) will perform the the search described above.

(LCL . @) Same as LC except the search is confined to the current expression, i.e., the edit chain is rebound during the search so that it looks as though the editor were called on just the current expression. For example, to find a COND containing a RETURN, one might use the location specification (COND (LCL RETURN) \) where the \ would reverse the effects of the LCL command, and make the final current expression be the COND.

(2ND . @) Same as (LC . @) followed by another (LC . @) except that if the first succeeds and second fails, no change is made to the edit chain.

(3RD . @) Similar to 2ND.

(← pattern) ascends the edit chain looking for a link which matches pattern. In other words, it keeps doing 0's until it gets to a specified point. If pattern is atomic, it is matched with the first element of each link, otherwise with the entire link.[27]

---

[27] If pattern is of the form (IF expression), expression is evaluated at each link, and if its value is NIL, or the evaluation causes an error, the ascent continues.

For example:

```
*PP
  [PROG NIL
        (COND
          [(NULL (SETQ L (CDR L)))
            (COND
              (FLG (RETURN L]
          ([NULL (CDR (FMEMB (CAR L)
                             (CADR L]]
*F CADR
*(← COND)
*P
(COND (& &) (& &))
*
```

Note that this command differs from BF in that it does not search *inside* of each link, it simply ascends. Thus in the above example, F CADR followed by BF COND would find (COND (FLG (RETURN L))), not the higher COND.

If no match is found, an error is generated, and the edit chain is unchanged.

(BELOW com x) — ascends the edit chain looking for a link specified by com, and stops x[28] links below that,[29] i.e. BELOW keeps doing 0's until it gets to a specified point, and then backs off x 0's.

(BELOW com) — same as (BELOW com 1).

For example, (BELOW COND) will cause the cond *clause* containing the current expression to become the new current expression. Thus if the current expression is as shown above, F CADR followed by (BELOW COND) will make the new

---

[28] x is evaluated, e.g., (BELOW com (IPLUS X Y).

[29] Only links that are elements are counted, not tails.

expression be ([NULL (CDR (FMEMB (CAR L) (CADR L] (GO LP)), and is therefore equivalent to 0 0 0 0.

The BELOW command is useful for locating a substructure by specifying something it contains. For example, suppose the user is editing a list of lists, and wants to find a sublist that contains a FOO (at any depth). He simply executes F FOO (BELOW \).

(NEX x) same as (BELOW x) followed by NX.

For example, if the user is deep inside of a SELECTQ clause, he can advance to the next clause with (NEX SELECTQ).

NEX same as (NEX ←).

The atomic form of NEX is useful if the user will be performing repeated executions of (NEX x). By simply MARKing (see page 9.34) the chain corresponding to x, he can use NEX to step through the sublists.

(NTH x) generalized NTH command. Effectively performs (LCL . x), followed by (BELOW \), followed by UP.

In other words, NTH locates x, using a search restricted to the current expression, and then backs up to the current level, where the new current expression is the tail whose first element contains, however deeply, the expression that was the terminus of the location operation. For example:

```
*p
(PROG (& &) LP (COND & &) (EDITCOM &) (SETQ UNFIND UF) (RETURN L))
*(NTH UF)
*p
... (SETQ UNFIND UF) (RETURN L))
*
```

If the search is unsuccessful, NTH generates an error and the edit chain is not changed.

Note that (NTH n) is just a special case of (NTH x), and in fact, no special check is made for x a number; both commands are executed identically.

(pattern .. @)[30] e.g., (COND .. RETURN). Finds a cond that contains a return, at any depth. Equivalent to (but more efficient than) (F pattern N), (LCL . @) followed by (← pattern).

For example, if the current expression is (PROG NIL [COND ((NULL L) (COND (FLG (RETURN L] --), then (COND .. RETURN) will make (COND (FLG (RETURN L))) be the current expression. Note that it is the innermost COND that is found, because this is the first COND encountered when ascending from the RETURN. In other words, (pattern .. @) is not *always* equivalent to (F pattern N), followed by (LCL . @) followed by \.

Note that @ is a location specification, not just a pattern. Thus (RETURN .. COND 2 3) can be used to find the RETURN which contains a COND whose first clause contains (at least) three elements. Note also that since @ permits any edit command, the user can write commands of the form (COND .. (RETURN .. COND)), which will locate the first COND that contains a RETURN that contains a COND.

---

30 An infix command, '..' is not a meta-symbol, it *is* the name of the command. @ is cddr of the command.

9.3.3 Commands That Save and Restore The Edit Chain

Several facilities are available for saving the current edit chain and later retrieving it: MARK, which marks the current chain for future reference, ←,[31] which returns to the last mark without destroying it, and ←←, which returns to the last mark and also erases it.

MARK — adds the current edit chain to the front of the list marklst.

← — makes the new edit chain be (CAR MARKLST). Generates an error if marklst is NIL, i.e., no MARKs have been performed, or all have been erased.

←← — similar to ← but also erases the MARK, i.e., performs (SETQ MARKLST (CDR MARKLST)).

Note that if the user has two chains marked, and wishes to return to the first chain, he must perform ←←, which removes the second mark, and then ←. However, the second mark is then no longer accessible. If the user wants to be able to return to either of two (or more) chains, he can use the following generalized MARK:

(MARK atom) — sets atom to the current edit chain,

(\ atom) — makes the current edit chain become the value of atom.

---

[31] An atomic command; do not confuse ← with the list command (← pattern).

If the user did not prepare in advance for returning to a particular edit chain, he may still be able to return to that chain with a single command by using \ or \P.

\	makes the edit chain be the value of unfind. Generates an error if unfind=NIL.

unfind is set to the current edit chain by each command that makes a "big jump", i.e., a command that usually performs more than a single ascent or descent, namely ↑, ←, ←←, !NX, all commands that involve a search, e.g., F, LC, .., BELOW, et al and \ and \P themselves.[32]

For example, if the user types F COND, and then F CAR, \ would take him back to the COND. Another \ would take him back to the CAR, etc.

\P	restores the edit chain to its state as of the last print operation, i.e. P, ?, or PP. If the edit chain has not changed since the last printing, \P restores it to its state as of the printing before that one, i.e., two chains are always saved.

For example, if the user types P followed by 3 2 1 P, \P will return to the first P, i.e., would be equivalent to 0 0 0.[33] Another \P would then take him back to the second P, i.e., the user could use \P to flip back and forth between the two edit chains.

---

[32] Except that unfind is not reset when the current edit chain is the top level expression, since this could always be returned to via the ↑ command.

[33] Note that if the user had typed P followed by F COND, he could use *either* \ or \P to return to the P, i.e., the action of \ and \P are independent.

(S var . @)

Sets var (using setq) to the current expression after performing (LC . @). Edit chain is not changed.

Thus (S FOO) will set foo to the current expression, (S FOO -1 1) will set foo to the first element in the last element of the current expression.

This ends the section on "Attention Changing Commands."

9.4 Commands That Modify Structure

The basic structure modification commands in the editor are:

(n)

$n \geq 1$ deletes the corresponding element from the current expression.

(n e_1 ... e_m)

$n,m \geq 1$ replaces the nth element in the current expression with e_1 ... e_m.

(-n e_1 ... e_m)

$n,m \geq 1$ inserts e_1 ... e_m before the nth element in the current expression.

(N e_1 ... e_m)

$m \geq 1$ attaches e_1 ... e_m at the end of the current expression.

As mentioned earlier:

all structure modification done by the editor is destructive, i.e. the editor uses rplaca and rplacd to physically change the structure it was given.

However, all structure modification is undoable, see UNDO page 9.78.

All of the above commands generate errors if the current expression is not a list, or in the case of the first three commands, if the list contains fewer than n elements. In addition, the command (1), i.e. delete the first element, will cause an error if there is only one element, since deleting the first element must be done by replacing it with the second element, and then deleting the second element. Or, to look at it another way, deleting the first element when there is only one element would require changing a list to an atom (i.e. to NIL) which cannot be done.[34]

9.4.1 Implementation of Structure Modification Commands

Note: Since all commands that insert, replace, delete or attach structure use the same low level editor functions, the remarks made here are valid for all structure changing commands.

For all replacement, insertion, and attaching at the end of a list, unless the command was typed in directly to the editor,[35] *copies* of the corresponding structure are used, because of the possibility that the exact same command, (i.e. same list structure) might be used again. Thus if a program constructs the command (1 (A B C)) e.g. via (LIST 1 FOO), and gives this command to the editor, the (A B C) used for the replacement will *not* be eq to foo.[36]

---

[34] However, the command DELETE will work even if there is only one element in the current expression, since it will ascend to a point where it *can* do the deletion.

[35] Some editor commands take as arguments a list of edit commands, e.g. (LP F FOO (1 (CAR FOO))). In this case, the command (1 (CAR FOO)) is not considered to have been "typed in" even though the LP command itself may have been typed in. Similarly, commands originating from macros, or commands given to the editor as arguments to editf, editv, et al, e.g. EDITF(FOO F COND (N --)) are not considered typed in.

[36] The user can circumvent this by using the I command, which computes the structure to be used. In the above example, the form of the command would be (I 1 FOO), which would replace the first element with the value of foo itself. See page 9.62.

The rest of this section is included for applications wherein the editor is used to modify a data structure, and pointers into that data structure are stored elsewhere. In these cases, the actual mechanics of structure modification must be known in order to predict the effect that various commands may have on these outside pointers. For example, if the value of *foo* is *cdr* of the current expression, what will the commands (2), (3), (2 X Y Z), (-2 X Y Z), etc. do to *foo*?

Deletion of the first element in the current expression is performed by replacing it with the second element and deleting the second element by patching around it. Deletion of any other element is done by patching around it, i.e., the previous tail is altered. Thus if *foo* is *eq* to the current expression which is (A B C D), and *fie* is *cdr* of *foo*, after executing the command (1), *foo* will be (B C D) (which is *equal* but not *eq* to *fie*). However, under the same initial conditions, after executing (2) *fie* will be unchanged, i.e., *fie* will still be (B C D) even though the current expression and *foo* are now (A C D).[37]

Both replacement and insertion are accomplished by smashing both *car* and *cdr* of the corresponding tail. Thus, if *foo* were *eq* to the current expression, (A B C D), after (1 X Y Z), *foo* would be (X Y Z B C D). Similarly, if *foo* were *eq* to the current expression, (A B C D), then after (-1 X Y Z), *foo* would be (X Y Z A B C D).

The N command is accomplished by smashing the last *cdr* of the current

---

[37] A general solution of the problem just isn't possible, as it would require being able to make two lists *eq* to each other that were originally different. Thus if *fie* is *cdr* of the current expression, and *fum* is *cddr* of the current expression, performing (2) would have to make *fie* be *eq* to *fum* if all subsequent operations were to update both *fie* and *fum* correctly. Think about it.

expression a la nconc. Thus if foo were eq to any tail of the current expression, after executing an N command, the corresponding expressions would also appear at the end of foo.

In summary, the only situation in which an edit operation will *not* change an external pointer occurs when the external pointer is to a *proper tail* of the data structure, i.e., to cdr of some node in the structure, and the operation is deletion. If all external pointers are to *elements* of the structure, i.e., to car of some node, or if only insertions, replacements, or attachments are performed, the edit operation will *always* have the same effect on an external pointer as it does on the current expression.

9.4.2 The A, B, and : Commands

In the (n), (n e_1 ... e_m), and (-n e_1 ... e_m) commands, the sign of the integer is used to indicate the operation. As a result, there is no direct way to express insertion after a particular element, (hence the necessity for a separate N command). Similarly, the user cannot specify deletion or replacement of the nth element from the end of a list without first converting n to the corresponding positive integer. Accordingly, we have:

(B e_1 ... e_m) — inserts e_1 ... e_m before the current expression. Equivalent to UP followed by (-1 e_1 ... e_m).

For example, to insert FOO before the last element in the current expression, perform -1 and then (B FOO).

(A e_1 ... e_m) — inserts e_1 ... e_m after the current expression. Equivalent to UP followed by (-2 e_1 ... e_m) or (N e_1 ... e_m) whichever is appropriate.

(: e_1 ... e_m) replaces the current expression by e_1 ... e_m. Equivalent to UP followed by (1 e_1 ... e_m).

DELETE, :, or (:) deletes the current expression.

DELETE first tries to delete the current expression by performing an UP and then a (1). This works in most cases. However, if after performing UP, the new current expression contains only one element, the command (1) will not work. Therefore, DELETE starts over and performs a BK, followed by UP, followed by (2). For example, if the current expression is (COND ((MEMB X Y)) (T Y)), and the user performs -1, and then DELETE, the BK-UP-(2) method is used, and the new current expression will be ... ((MEMB X Y)))

However, if the next higher expression contains only one element, BK will not work. So in this case, DELETE performs UP, followed by (: NIL), i.e., it *replaces* the higher expression by NIL. For example, if the current expression is (COND ((MEMB X Y)) (T Y)) and the user performs F MEMB and then DELETE, the new current expression will be ... NIL (T Y)) and the original expression would now be (COND NIL (T Y)). The rationale behind this is that deleting (MEMB X Y) from ((MEMB X Y)) changes a list of one element to a list of no elements, i.e., () or NIL.

If the current expression is a tail, then B, A, :, and DELETE all work exactly the same as though the current expression were the first element in that tail. Thus if the current expression were ... (PRINT Y) (PRINT Z)), (B (PRINT X)) would insert (PRINT X) before (PRINT Y), leaving the current expression ... (PRINT X) (PRINT Y) (PRINT Z)).

The following forms of the A, B, and : commands incorporate a location specification:

(INSERT e_1 ... e_m BEFORE . @)[38] Similar to (LC .@)[39] followed by (B e_1 ... e_m).

```
*P
(PROG (& & X) **COMMENT** (SELECTQ ATM & NIL) (OR & &) (PRIN1 & T)
(PRIN1 & T) (SETQ X &

*(INSERT LABEL BEFORE PRIN1)
*P
(PROG (& & X) **COMMENT** (SELECTQ ATM & NIL) (OR & &) LABEL
(PRIN1 & T) (                          40
*
```

Current edit chain is not changed, but unfind is set to the edit chain after the B was performed, i.e. \ will make the edit chain be that chain where the insertion was performed.

(INSERT e_1 ... e_m AFTER . @) Similar to INSERT BEFORE except uses A instead of B.

(INSERT e_1 ... e_m FOR . @) similar to INSERT BEFORE except uses : for B.

---

[38] i.e. @ is cdr[member[BEFORE;command]]

[39] except that if @ causes an error, the location process does *not* continue as described on page 9.29. For example if @=(COND 3) and the next COND does not have a 3rd element, the search stops and the INSERT fails. Note that the user can always write (LC COND 3) if he intends the search to continue.

[40] Sudden termination of output followed by a blank line return indicates printing was aborted by control-E.

(REPLACE @ WITH e_1 ... e_m)[41] Here @[42] is the *segment* of the command between REPLACE and WITH. Same as (INSERT e_1 ... e_m FOR . @).

Example: (REPLACE COND -1 WITH (T (RETURN L)))

(CHANGE @ TO e_1 ... e_m) Same as REPLACE WITH.

(DELETE . @) does a (LC . @)[43] followed by DELETE. Current edit chain is not changed,[44] but unfind is set to the edit chain after the DELETE was performed.

Example: (DELETE -1), (DELETE COND 3)

Note: if @ is NIL (i.e. empty), the corresponding operation is performed here *(on the current edit chain).*

For example, (REPLACE WITH (CAR X)) is equivalent to (: (CAR X)). For added readability, HERE is also permitted, e.g. (INSERT (PRINT X) BEFORE HERE) will insert (PRINT X). before the current expression (but not change the edit chain).

---

41 BY can be used for WITH.

42 See footnote on page 9.41.

43 See footnote on page 9.41.

44 Unless the current expression is no longer a part of the expression being edited, e.g. if the current expression is ... C) and the user performs (DELETE 1), the tail, (C), will have been cut off. Similarly, if the current expression is (CDR Y) and the user performs (REPLACE WITH (CAR X)).

Note: @ does not have to specify a location within the current expression, i.e. it is perfectly legal to ascend to INSERT, REPLACE, or DELETE

For example, (INSERT (RETURN) AFTER ↑ PROG -1) will go to the top, find the first PROG, and insert a (RETURN) at its end, and not change the current edit chain.

The A, B, and : commands, commands, (and consequently INSERT, REPLACE, and CHANGE), all make special checks in e_1 thru e_m for expressions of the form (## . coms). In this case, the expression used for inserting or replacing is a *copy* of the current expression after executing coms, a list of edit commands.[45] For example, (INSERT (## F COND -1 -1) AFTER 3)[46] will make a copy of the last form in the last clause of the next cond, and insert it after the third element of the current expression.

9.4.3 Form Oriented Editing and the Role of UP

The UP that is performed before A, B, and : commands[47] makes these operations form-oriented. For example, if the user types F SETQ, and then DELETE, or simply (DELETE SETQ), he will delete the entire SETQ expression, whereas (DELETE X) if X is a variable, deletes just the variable X. In both cases, the operation is performed on the corresponding *form*, and in both cases is probably what the user intended. Similarly, if the user types (INSERT (RETURN Y) BEFORE SETQ), he means before the SETQ expression, not

---

[45] The execution of coms does not change the current edit chain.

[46] *Not* (INSERT F COND -1 (## -1) AFTER 3), which inserts four elements after the third element, namely F, COND, -1, and a copy of the last element in the current expression.

[47] and therefore in INSERT, CHANGE, REPLACE, and DELETE commands after the location portion of the operation has been performed.

before the atom SETQ.[48] A consequent of this procedure is that a pattern of the form (SETQ Y --) can be viewed as simply an elaboration and further refinement of the pattern SETQ. Thus (INSERT (RETURN Y) BEFORE SETQ) and (INSERT (RETURN Y) BEFORE (SETQ Y --)) perform the same operation[49] and, in fact, this is one of the motivations behind making the current expression after F SETQ, and F (SETQ Y --) be the same.

Occasionally, however, a user may have a data structure in which no special significance or meaning is attached to the position of an atom in a list, as INTERLISP attaches to atoms that appear as car of a list, versus those appearing elsewhere in a list. In general, the user may not even *know* whether a particular atom is at the head of a list or not. Thus, when he writes (INSERT expression BEFORE FOO), he means before the atom FOO, whether or not it is car of a list. By setting the variable upfindflg to NIL,[50] the user can suppress the implicit UP that follows searches for atoms, and thus achieve the desired effect. With upfindflg=NIL, following F FOO, for example, the current expression will be the atom FOO. In this case, the A, B, and : operations will operate with respect to the atom FOO. If the user intends the operation to refer to the list which FOO heads, he simply uses instead the pattern (FOO --).

---

[48] There is some ambiguity in (INSERT expr AFTER functionname), as the user might mean make expr be the function's first argument. Similarly, the user cannot write (REPLACE SETQ WITH SETQQ) meaning change the name of the function. The user must in these cases write (INSERT expr AFTER functioname 1), and (REPLACE SETQ 1 WITH SETQQ).

[49] assuming the next SETQ is of the form (SETQ Y --).

[50] Initially, and usually, set to T.

9.4.4 Extract and Embed

Extraction involves replacing the current expression with one of its subexpressions (from any depth).

(XTR . @) — replaces the original current expression with the expression that is current after performing (LCL . @).[51]

For example, if the current expression is (COND ((NULL X) (PRINT Y))), (XTR PRINT), or (XTR 2 2) will replace the cond by the print.

If the current expression after (LCL . @) is a *tail* of a higher expression, its first element is used.

For example, if the current expression is (COND ((NULL X) Y) (T Z)), then (XTR Y) will replace the cond with Y, even though the current expression after performing (LCL Y) is ... Y).

If the extracted expression is a list, then after XTR has finished, the current expression will be that list.

Thus, in the first example, the current expression after the XTR would be (PRINT Y).

---

[51] See footnote on page 9.41.

If the extracted expression is not a list, the new current expression will be a tail whose first element is that non-list.

Thus, in the second example, the current expression after the XTR would be ... Y followed by whatever followed the COND.

If the current expression *initially* is a tail, extraction works exactly the same as though the current expression were the first element in that tail. Thus if the current expression is ... (COND ((NULL X) (PRINT Y))) (RETURN Z)), then (XTR PRINT) will replace the cond by the print, leaving (PRINT Y) as the current expression.

The extract command can also incorporate a location specification:

(EXTRACT $@_1$ FROM . $@_2$)[52] — Performs (LC . $@_2$)[53] and then (XTR . $@_1$). Current edit chain is not changed, but unfind is set to the edit chain after the XTR was performed.

Example: If the current expression is (PRINT (COND ((NULL X) Y) (T Z))) then following (EXTRACT Y FROM COND), the current expression will be (PRINT Y). (EXTRACT 2 -1 FROM COND), (EXTRACT Y FROM 2), (EXTRACT 2 -1 FROM 2) will all produce the same result.

---

[52] $@_1$ is the *segment* between EXTRACT and FROM.

[53] See footnote on page 9.41.

* * *

While extracting replaces the current expression by a subexpression, embedding replaces the current expression with one containing *it* as a subexpression.

(MBD e_1 ... e_m) — MBD substitutes[54] the current expression for all instances of the atom * in e_1 ... e_m, and replaces the current expression with the result of that substitution.

Examples: If the current expression is (PRINT Y),
(MBD (COND ((NULL X) *) ((NULL (CAR Y)) * (GO LP)))) would replace (PRINT Y) with (COND ((NULL X) (PRINT Y)) ((NULL (CAR Y)) (PRINT Y) (GO LP))).

If the current expression is (RETURN X), (MBD (PRINT Y) (AND FLG *)) would replace it with the *two* expressions (PRINT Y) and (AND FLG (RETURN X)) i.e., if the (RETURN X) appeared in the cond clause (T (RETURN X)), after the MBD, the clause would be (T (PRINT Y) (AND FLG (RETURN X))).

If * does not appear in e_1 ... e_m, the MBD is interpreted as (MBD (e_1 ... e_m *)).

Examples: If the current expression is (PRINT Y), then (MBD SETQ X) will replace it with (SETQ X (PRINT Y)). If the current expression is (PRINT Y), (MBD RETURN) will replace it with (RETURN (PRINT Y)).

MBD leaves the edit chain so that the larger expression is the new current expression.

---

[54] as with <u>subst</u>, a fresh copy is used for each substitution.

If the current expression *initially* is a tail, embedding works exactly the same as though the current expression were the first element in that tail. Thus if the current expression were ... (PRINT Y) (PRINT Z)), (MBD SETQ X) would replace (PRINT Y) with (SETQ X (PRINT Y)).

The embed command can also incorporate a location specification:

(EMBED @ IN . x)[55] — does (LC . @)[56] and then (MBD . x). Edit chain is not changed, but unfind is set to the edit chain after the MBD was performed.

Example: (EMBED PRINT IN SETQ X), (EMBED 3 2 IN RETURN), (EMBED COND 3 1 IN (OR * (NULL X))).

WITH can be used for IN, and SURROUND can be used for EMBED, e.g., (SURROUND NUMBERP WITH (AND * (MINUSP X))).

9.4.5 The MOVE Command

The MOVE command allows the user to specify (1) the expression to be moved, (2) the place it is to be moved to, and (3) the operation to be performed there, e.g., insert it before, insert it after, replace, etc.

(MOVE $@_1$ TO com . $@_2$)[57] — where com is BEFORE, AFTER, or the name of a list

---

[55] @ is the segment between EMBED and IN.

[56] See footnote on page 9.41.

[57] $@_1$ is the segment between MOVE and TO.

command, e.g., :, N, etc. performs (LC . $@_1$),[58] and obtains the current expression there (or its first element, if it is a tail), which we will call expr; MOVE then goes back to the original edit chain, performs (LC . $@_2$) followed by (com expr),[59] then goes back to $@_1$ and deletes expr. Edit chain is not changed. Unfind is set to edit chain after (com expr) was performed.

For example, if the current expression is (A B C D), (MOVE 2 TO AFTER 4) will make the new current expression be (A C D B). Note that 4 was executed as of the original edit chain, and that the second element had not yet been removed.[60]

As the following examples taken from actual editing will show, the MOVE command is an extremely versatile and powerful feature of the editor.

```
*?
(PROG ((L L)) (EDLOC (CDDR C)) (RETURN (CAR L)))
*(MOVE 3 TO : CAR)
*?
(PROG ((L L)) (RETURN (EDLOC (CDDR C))))
*

*p
... (SELECTQ OBJPR & &) (RETURN &) LP2 (COND & &))
*(MOVE 2 TO N 1)
*p
... (SELECTQ OBJPR & & &) LP2 (COND & &))
*
```

---

[58] see footnote on page 9.41.

[59] Setting an internal flag so expr is not copied.

[60] If $@_2$ specifies a location *inside of the expression to be moved*, a message is printed and an error is generated, e.g. (MOVE 2 TO AFTER X), where X is contained inside of the second element.

```
*p
(OR (EQ X LASTAIL) (NOT &) (AND & & &))
*(MOVE 4 TO AFTER (BELOW COND))
*p
(OR (EQ X LASTAIL) (NOT &))
*\ p
... (& &) (AND & & &) (T & &))
*
```

```
*p
((NULL X) **COMMENT** (COND & &))
*(-3 (GO NXT]
*(MOVE 4 TO N (← PROG))
*p
((NULL X) **COMMENT** (GO NXT))
*\ p
(PROG (&) **COMMENT** (COND & & &) (COND & & &) (COND & &))
*(INSERT NXT BEFORE -1)
*p
(PROG (&) **COMMENT** (COND & & &) (COND & & &) NXT (COND & &))
```

Note that in the last example, the user could have added the prog label NXT and moved the cond in one operation by performing (MOVE 4 TO N (← PROG) (N NXT)). Similarly, in the next example, in the course of specifying $@_2$, the location where the expression was to be moved to, the user also performs a structure modification, via (N (T)), thus creating the structure that will receive the expression being moved.

```
*p
((CDR &) **COMMENT** (SETQ CL &) (EDITSMASH CL & &))
*MOVE 4 TO N 0 (N (T)) -1]
*p
((CDR &) **COMMENT** (SETQ CL &))
*\ p
*(T (EDITSMASH CL & &))
*
```

If $@_2$ is NIL, or (HERE), the current position specifies where the operation is to take place. In this case, unfind is set to where the expression that was moved was originally located, i.e. $@_1$. For example:

```
*p
(TENEX)
*(MOVE ↑ F APPLY TO N HERE)
*p
(TENEX (APPLY & &))
*
```

```
*p
(PROG (& & & ATM IND VAL) (OR & &)   **COMMENT**   (OR & &) (PRIN1 & T) (
PRIN1 & T) (SETQ IND
```
[61]

```
*(MOVE * TO BEFORE HERE)
*p
(PROG (& & & ATM IND VAL) (OR & &) (OR & &) (PRIN1 &

*p
(T (PRIN1 C-EXP T))
*(MOVE ↑ BF PRIN1 TO N HERE)
*p
(T (PRIN1 C-EXP T) (PRIN1 & T))
*
```

Finally, if $@_1$ is NIL, the MOVE command allows the user to specify where the *current expression* is to be moved to. In this case, the edit chain is changed, and is the chain where the current expression was moved to; unfind is set to where it was.

```
*p
(SELECTQ OBJPR (&) (PROGN & &))
*(MOVE TO BEFORE LOOP)
*p
... (SELECTQ OBJPR & &) LOOP (FRPLACA DFPRP &) (FRPLACD DFPRP
&) (SELECTQ
*
```

9.4.6 Commands That "Move Parentheses"

The commands presented in this section permit modification of the list structure itself, as opposed to modifying components thereof. Their effect can be described as inserting or removing a single left or right parenthesis, or pair of left and right parentheses. Of course, there will always be the same number of left parentheses as right parentheses in any list structure, since the parentheses are just a notational guide to the structure provided by print. Thus, no command can insert or remove just one parenthesis, but this is suggestive of what actually happens.

---

[61] Sudden termination of output followed by a blank line indicates printing was aborted by control-E.

In all six commands, n and m are used to specify an element of a list, usually of the current expression. In practice, n and m are usually positive or negative integers with the obvious interpretation. However, all six commands use the generalized NTH command, page 9.32, to find their element(s), so that nth element means the first element of the tail found by performing (NTH n). In other words, if the current expression is (LIST (CAR X) (SETQ Y (CONS W Z))), then (BI 2 CONS), (BI X -1), and (BI X Z) all specify the exact same operation.

All six commands generate an error if the element is not found, i.e. the NTH fails. All are undoable.

(BI n m) both in, inserts a left parentheses before the nth element and after the mth element in the current expression. Generates an error if the mth element is not contained in the nth tail, i.e., the mth element must be "to the right" of the nth element.

Example: If the current expression is (A B (C D E) F G), then (BI 2 4) will modify it to be (A (B (C D E) F) G).

(BI n) same as (BI n n).

Example: If the current expression is (A B (C D E) F G), then (BI -2) will modify it to be (A B (C D E) (F) G).

(BO n) both out. Removes both parentheses from the nth element. Generates an error if nth element is not a list.

Example: If the current expression is (A B (C D E) F G), then (BO D) will modify it to be (A B C D E F G).

(LI n) left in, inserts a left parenthesis before the nth element (and a matching right parenthesis at the end of the current expression), i.e. equivalent to (BI n -1).

Example: if the current expression is (A B (C D E) F G), then (LI 2) will modify it to be (A (B (C D E) F G)).

(LO n) left out, removes a left parenthesis from the nth element. *All elements following the nth element are deleted*. Generates an error if nth element is not a list.

Example: If the current expression is (A B (C D E) F G), then (LO 3) will modify it to be (A B C D E).

(RI n m) right in, inserts a right parenthesis after the mth element of the nth element. The rest of the nth element is brought up to the level of the current expression.

Example: If the current expression is (A (B C D E) F G), (RI 2 2) will modify it to be (A (B C) D E F G). Another way of thinking about RI is to read it as "move the right parenthesis at the end of the nth element *in* to after its mth element."

(RO n) right out, removes the right parenthesis from the nth element, moving it to the end of the current expression. All elements following the nth element are moved inside of the nth element. Generates an error if nth element is not a list.

Example: If the current expression is (A B (C D E) F G), (RO 3) will modify it to be (A B (C D E F G)). Another way of thinking about RO is to read it as "move the right parenthesis at the end of the nth element *out* to the end of the current expression."

9.4.7 TO and THRU

EXTRACT, EMBED, DELETE, REPLACE, and MOVE can be made to operate on several contiguous elements, i.e., a segment of a list, by using in their respective location specifications the TO or THRU command.

($@_1$ THRU $@_2$) — does a (LC . $@_1$), followed by an UP, and then a (BI 1 $@_2$), thereby grouping the segment into a single element, and finally does a 1, making the final current expression be that element.

For example, if the current expression is (A (B (C D) (E) (F G H) I) J K), following (C THRU G), the current expression will be ((C D) (E) (F G H)).

($@_1$ TO $@_2$) — Same as THRU except last element not included, i.e., after the BI, an (RI 1 -2) is performed.

If both $@_1$ and $@_2$ are numbers, and $@_2$ is greater than $@_1$, then $@_2$ counts from the beginning of the current expression, the same as $@_1$. In other words, if the current expression is (A B C D E F G), (3 THRU 5) means (C THRU E) not (C THRU G). In this case, the corresponding BI command is (BI 1 $@_2$-$@_1$+1).

THRU and TO are not very useful commands by themselves; they are intended to be used in conjunction with EXTRACT, EMBED, DELETE, REPLACE, and MOVE. After THRU and TO have operated, they set an internal editor flag informing the above

commands that the element they are operating on is actually a segment, and that the extra pair of parentheses should be removed when the operation is complete. Thus:

```
*p
(PROG (& & ATM IND VAL WORD) (PRIN1 & T) (PRIN1 & T) (SETQ IND &) (SETQ VAL &)
**COMMENT** (SETQQ

*(MOVE (3 THRU 4) TO BEFORE 7)
*p
(PROG (& & ATM IND VAL WORD) (SETQ IND &) (SETQ VAL &) (PRIN1 & T)  (PRIN1 & T)
**COMMENT**

*
```

```
*p
(* FAIL RETURN FROM EDITOR. USER SHOULD NOTE THE VALUES OF SOURCEXPR AND
CURRENTFORM.  CURRENTFORM IS THE LAST FORM IN SOURCEXPR WHICH WILL HAVE BEEN
TRANSLATED, AND IT CAUSED THE ERROR.)
*(DELETE (USER THRU CURR$))
=CURRENTFORM.
*p
(* FAIL RETURN FROM EDITOR.  CURRENTFORM IS

*
```

```
*p
... LP (SELECTQ & & & & NIL) (SETQ Y &) OUT (SETQ FLG &) (RETURN Y))
*(MOVE (1 TO OUT) TO N HERE]
*p
... OUT (SETQ FLG &) (RETURN Y) LP (SELECTQ & & & & NIL) (SETQ Y &))
*
```

```
*pp
  [PROG (RF TEMP1 TEMP2)
        (COND
          ((NOT (MEMB REMARG LISTING))
            (SETQ TEMP1 (ASSOC REMARG NAMEDREMARKS)) **COMMENT**
            (SETQ TEMP2 (CADR TEMP1))
            (GO SKIP))
           (T    **COMMENT**
            (SETQ TEMP1 REMARG)))
        (NCONC1 LISTING REMARG)
        (COND
           ((NOT (SETQ TEMP2 (SASSOC

*(EXTRACT (SETQ THRU CADR) FROM COND)
*p
(PROG (RF TEMP1 TEMP2) (SETQ TEMP1 &) **COMMENT** (SETQ TEMP2 &)
(NCONC1 LISTING REMARG) (COND & &

*
```

TO and THRU can also be used directly with XTR.[62] Thus in the previous example, if the current expression had been the COND, e.g. the user had first performed F COND, he could have used (XTR (SETQ THRU CADR)) to perform the extraction.

($@_1$ TO), ($@_1$ THRU) — both same as ($@_1$ THRU -1), i.e., from $@_1$ through the end of the list.

Examples:

```
*p
(VALUE (RPLACA DEPRP &) (RPLACD &) (RPLACA VARSWORD &) (RETURN))
*(MOVE (2 TO) TO N (← PROG))
*(N (GO VAR))
*p
(VALUE (GO VAR))
```

```
*p
(T **COMMENT** (COND &) **COMMENT** (EDITSMASH CL & &) (COND &))
*(-3 (GO REPLACE))
*(MOVE (COND TO) TO N ↑ PROG (N REPLACE))
*p
(T **COMMENT** (GO REPLACE))
*\ P
(PROG (&) **COMMENT** (COND & & &) (COND & & &) DELETE (COND & &)
REPLACE (COND &) **COMMENT** (EDITSMASH CL & &) (COND &))
*
```

---

62 Because XTR involves a location specification while A, B, :, and MBD do not.

```
*pp
  [LAMBDA (CLAUSALA X)
    (PROG (A D)
          (SETQ A CLAUSALA)
      LP  (COND
            ((NULL A)
              (RETURN)))
          (SERCH X A)
          (RUMARK (CDR A))
          (NOTICECL (CAR A))
          (SETQ A (CDR A))
          (GO LP]
*(EXTRACT (SERCH THRU NOT$) FROM PROG)
=NOTICECL
*p
(LAMBDA (CLAUSALA X) (SERCH X A) (RUMARK &) (NOTICECL &))
*(EMBED (SERCH TO) IN (MAP CLAUSALA (FUNCTION (LAMBDA (A) *]
*pp
  [LAMBDA (CLAUSALA X)
    (MAP CLAUSALA (FUNCTION (LAMBDA (A)
             (SERCH X A)
             (RUMARK (CDR A))
             (NOTICECL (CAR A]
*
```

9.4.8 The R Command

(R x y) — replaces all instances of x by y in the current expression, e.g., (R CAADR CADAR). Generates an error if there is not at least one instance.

The R command operates in conjunction with the search mechanism of the editor. The search proceeds as described on page 9.23-25, and x can employ any of the patterns on page 9.21-23. Each time x matches an element of the structure, the element is replaced by (a copy of) y; each time x matches a tail of the structure, the tail is replaced by (a copy of) y.

For example, if the current expression is (A (B C) (B . C)),
(R C D) will change it to (A (B D) (B . D)),
(R (... . C) D) to (A (B C) (B . D)),
(R C (D E)) to (A (B (D E)) (B D E)), and
(R (... . NIL) D) to (A (B C . D) (B . C) . D).

If x is an atom or string containing alt-modes, alt-modes appearing in y stand for the characters matched by the corresponding alt-mode in x. For example, (R FOO$ FIE$) means for all atoms or strings that begin with FOO, replace the characters 'FOO' by 'FIE'.[63] Applied to the list (FOO FOO2 XFOO1), (R FOO$ FIE$) would produce (FIE FIE2 XFOO1), and (R FOO FIE) would produce (FIE FIE2 XFIE1). Similarly, (R D A) will change (LIST (CADR X) (CADDR Y)) to (LIST (CAAR X) (CAADR)).[64]

The user will be informed of all such alt-mode replacements by a message of the form x->y, e.g. CADR->CAAR.

Note that the $ feature can be used to delete or add characters, as well as replace them. For example, (R $1 $) will delete the terminating 1's from all literal atoms and strings. Similarly, if an alt-mode in x does not have a mate in y, the characters matched by the $ are effectively deleted. For example, (R $/$ $) will change AND/OR to AND.[65] y can also be a list containing alt-modes, e.g. (R $1 (CAR $)) will change FOO1 to (CAR FOO), FIE1 to (CAR FIE).

If x does not contain alt-modes, $ appearing in y refers to the *entire*

---

[63] If x matches a string, it will be replaced by a string. Note that it does not matter whether x or y themselves are strings, i.e. (R D A), (R "D" A), (R D "A"), and (R "D" "A") are equivalent. Note also that x will never match with a number, i.e. (R $1 $2) will not change 11 to 12.

[64] Note that CADDR was *not* changed to CAAAR, i.e. (R D A) does not mean replace every D with A, but replace the first D in every atom or string by A. If the user wanted to replace every D by A, he could perform (LP (R D A)).

[65] However, there is no similar operation for changing AND/OR to OR, since the first $ in y always corresponds to the first $ in x, the second $ in y to the second in x, etc.

expression matched by x, e.g. (R LONGATOM '$) changes LONGATOM to 'LONGATOM, (R (SETQ X &) (PRINT $)) changes every (SETQ X &) to (PRINT (SETQ X &)).[66]

Since (R x y) is a frequently used operation for replacing characters, the following command is provided:

(RC x y) equivalent to (R x y)

R and RC change all instances of x to y. The commands R1 and RC1 are available for changing just one, (i.e. the first) instance of x to y.

(R1 x y) find the first instance of x and replace it by y.

(RC1 x y) (R1 x y).

In addition, while R and RC only operate within the current expression, R1 and RC1 will continue searching, a la the F command, until they find an instance of x, even if the search carries them beyond the current expression.

(SW n m) switches the nth and mth elements of the current expression.

For example, if the current expression is
(LIST (CONS (CAR X) (CAR Y)) (CONS (CDR X) (CDR Y))),
(SW 2 3) will modify it to be
(LIST (CONS (CDR X) (CDR Y)) (CONS (CAR X) (CAR Y))). The relative order of n and m is not important, i.e., (SW 3 2) and (SW 2 3) are equivalent.

---

[66] If x is a pattern containing an alt-mode pattern somewhere *within* it, the characters matched by the alt-modes are not available, and for the purposes of replacement, the effect is the same as though x did not contain any alt-modes. For example, if the user types (R (CAR F$) (PRINT $)), the second $ will refer to the entire expression matched by (CAR F$).

SW uses the generalized NTH command to find the *n*th and *m*th elements, a la the BI-BO commands.

Thus in the previous example, (SW CAR CDR) would produce the same result.

9.5 Commands That Print

PP — prettyprints the current expression.

P — prints the current expression as though *printlevel* were set to 2.

(P m) — prints *mth* element of current expression as though *printlevel* were set to 2.

(P 0) — same as P

(P m n) — prints *mth* element of current expression as though *printlevel* were set to *n*.

(P 0 n) — prints current expression as though *printlevel* were set to *n*.

? — same as (P 0 100)

Both (P m) and (P m n) use the generalized NTH command to obtain the corresponding element, so that *m* does not have to be a number, e.g. (P COND 3) will work. PP causes all comments to be printed as **COMMENT** (see Section

14). P and ? print as **COMMENT** only those comments that are (top level) elements of the current expression.[67]

PP* — prettyprints current expression, *including* comments.

PP* is equivalent to PP except that it first resets **comment**flg to NIL (see Section 14). In fact, it is defined as (RESETVAR **COMMENT**FLG NIL PP), see page 9.77.

PPV — prettyprints current expression as a variable, i.e. no special treatment for LAMBDA, COND, SETQ, etc., or for CLISP.

PPT — prettyprints current expression, printing CLISP translations, if any.

All printing functions print to the teletype, regardless of the primary output file. No printing function ever changes the edit chain. All record the current edit chain for use by \P, page 9.35. All can be aborted with control-E.

---

[67] Lower expressions are not really seen by the editor; the printing command simply sets printlevel and calls print.

9.6 Commands That Evaluate

E — *only when typed in*,[68] causes the editor to call lispx giving it the next input as argument.[69]

```
Example: *E BREAK(FIE FUM)
         (FIE FUM)
         *E (FOO)

              (FIE BROKEN)
         :
```

(E x) — evaluates x, i.e., performs eval[x], and prints the result on the teletype.

(E x T) — same as (E x) but does not print.

The (E x) and (E x T) commands are mainly intended for use by macros and subroutine calls to the editor; the user would probably type in a form for evaluation using the more convenient format of the (atomic) E command.

(I c x_1 ... x_n) — same as (C y_1 ... y_n) where y_i=eval[x_i].

Example: (I 3 (GETD (QUOTE FOO))) will replace the 3rd element of the current expression with the definition of foo.[70] (I N FOO (CAR FIE)) will attach the

---

[68] e.g, (INSERT D BEFORE E) will treat E as a pattern, and search for E.

[69] lispx is used by evalqt and break for processing teletype inputs. If nothing else is typed on the same line, lispx evaluates its argument. Otherwise, lispx applies it to the next input. In both cases, lispx prints the result. See above example, and Sections 2 and 22.

[70] The I command sets an internal flag to indicate to the structure modification commands *not* to copy expression(s) when inserting, replacing, or attaching.

value of foo and car of the value of fie to the end of the current expression. (I F= FOO T) will search for an expression eq to the value of foo.

If c is not an atom, c is evaluated also.

Example: (I (COND ((NULL FLG) (QUOTE -1)) (T 1)) FOO), if flg is NIL, inserts the value of foo before the first element of the current expression, otherwise replaces the first element by the value of foo.

##[com_1;com_2; ... ;com_n] is an NLAMBDA, NOSPREAD function (not a command). Its value is what the current expression would be after executing the edit commands com_1 ... com_n starting from the present edit chain. Generates an error if any of com_1 thru com_n cause errors. The current edit chain is never changed.[71]

Example: (I R (QUOTE X) (## (CONS .. Z))) replaces all X's in the current expression by the first cons containing a Z.

The I command is not very convenient for computing an *entire* edit command for execution, since it computes the command name and its arguments separately. Also, the I command cannot be used to compute an atomic command. The following two commands provide more general ways of computing commands.

(COMS x_1 ... x_n) Each x_i is evaluated and its value is executed as a command.

---

[71] Recall that A, B, :, INSERT, REPLACE, and CHANGE make special checks for ## forms in the expressions used for inserting or replacing, and use a copy of ## form instead (see page 9.43). Thus, (INSERT (## 3 2) AFTER 1) is equivalent to (I INSERT (COPY (## 3 2)) (QUOTE AFTER) 1).

For example, (COMS (COND (X (LIST 1 X)))) will replace the first element of the current expression with the value of x if non-NIL, otherwise do nothing.[72]

(COMSQ com_1 ... com_n)   executes com_1 ... com_n.

COMSQ is mainly useful in conjunction with the COMS command. For example, suppose the user wishes to compute an entire list of commands for evaluation, as opposed to computing each command one at a time as does the COMS command. He would then write (COMS (CONS (QUOTE COMSQ) x)) where x computed the list of commands, e.g., (COMS (CONS (QUOTE COMSQ) (GETP FOO (QUOTE COMMANDS)))).

9.7 Commands That Test

(IF x)   generates an error *unless* the value of eval[x] is true, i.e., if eval[x] causes an error or eval[x]=NIL, IF will cause an error.

For some editor commands, the occurrence of an error has a well defined meaning, i.e., they use errors to branch on, as cond uses NIL and non-NIL. For example, an error condition in a location specification may simply mean "not this one, try the next." Thus the location specification (IPLUS (E (OR (NUMBERP (## 3)) (ERROR!)) T)) specifies the first IPLUS whose second argument is a number. The IF command, by equating NIL to error, provides a more natural way of accomplishing the same result. Thus, an equivalent location specification is (IPLUS (IF (NUMBERP (## 3)))).

---

[72] because NIL as a command is a NOP, see page 9.70.

The IF command can also be used to select between two alternate lists of commands for execution.

(IF x $coms_1$ $coms_2$) — If eval[x] is true, execute $coms_1$; if eval[x] causes an error or is equal to NIL, execute $coms_2$.[73]

For example, the command (IF (READP T) NIL (P)) will print the current expression provided the input buffer is empty.

IF can also be written as:

(IF x $coms_1$) — if eval[x] is true, execute $coms_1$; otherwise generate an error.

(LP . coms) — repeatedly executes coms, a list of commands, until an error occurs.

For example, (LP F PRINT (N T)) will attach a T at the end of every print expression. (LP F PRINT (IF (## 3) NIL ((N T)))) will attach a T at the end of each print expression which does not already have a second argument.[74]

When an error occurs, LP prints n OCCURRENCES.

---

[73] Thus IF is equivalent to (COMS (CONS (QUOTE COMSQ) (COND
((CAR (NLSETQ (EVAL X))) COMS1)
(T COMS2)))).

[74] i.e. the form (## 3) will cause an error if the edit command 3 causes an error, thereby selecting ((N T)) as the list of commands to be executed. The IF could also be written as (IF (CDDR (##)) NIL ((N T))).

where n is the number of times <u>coms</u> was successfully executed. The edit chain is left as of the last complete successful execution of <u>coms</u>.

(LPQ . coms) same as LP but does not print the message n OCCURRENCES.

In order to prevent non-terminating loops, both LP and LPQ terminate when the number of iterations reaches <u>maxloop</u>, initially set to 30.[75] Since the edit chain is left as of the last successful completion of the loop, the user can simply continue the LP command with REDO (Section 22).

(SHOW . x) <u>x</u> is a list of patterns. SHOW does a LPQ printing all instances of the indicated expression(s), e.g. (SHOW FOO (SETQ FIE &)) will print all FOO's and all (SETQ FIE &)'s. Generates an error if there aren't any instances of the expression(s).

(EXAM . x) like SHOW except calls the editor recursively (via the TTY: command described on page 9.70) on each instance of the indicated espression(s) so that the user can examine and/or change them.

(ORR $coms_1$ ... $coms_n$) ORR begins by executing $coms_1$, a list of commands. If no error occurs, ORR is finished. Otherwise, ORR restores the edit chain to its original value, and continues by executing $coms_2$, etc. If none of the command lists execute without errors, i.e.,

---

75 <u>maxloop</u> can also be set to NIL, which is equivalent to infinity.

the ORR "drops off the end", ORR generates an error. Otherwise, the edit chain is left as of the completion of the first command list which executes without an error.[76]

For example, (ORR (NX) (!NX) NIL) will perform a NX, if possible, otherwise a !NX, if possible, otherwise do nothing. Similarly, DELETE could be written as (ORR (UP (1)) (BK UP (2)) (UP (: NIL))).

9.8 Macros

Many of the more sophisticated branching commands in the editor, such as ORR, IF, etc., are most often used in conjunction with edit macros. The macro feature permits the user to define new commands and thereby expand the editor's repertoire.[77] Macros are defined by using the M command.

(M c . coms) — For c an atom, M defines c as an atomic command.[78] Executing c is then the same as executing the list of commands coms.

For example, (M BP BK UP P) will define BP as an atomic command which does three things, a BK, and UP, and a P. Macros can use commands defined by macros

---

[76] NIL as a command list is perfectly legal, and will always execute successfully. Thus, making the last 'argument' to ORR be NIL will insure that the ORR never causes an error. Any other atom is treated as (atom), i.e., the above example could be written as (OR NX !NX NIL).

[77] However built in commands always take precedence over macros, i.e., the editor's repertoire can be expanded, but not redefined.

[78] If a macro is redefined, its new definition replaces its old.

as well as built in commands in their definitions. For example, suppose Z is defined by (M Z -1 (IF (READP T) NIL (P))), i.e. Z does a -1, and then if nothing has been typed, a P. Now we can define ZZ by (M ZZ -1 Z), and ZZZ by (M ZZZ -1 -1 Z) or (M ZZZ -1 ZZ).

Macros can also define list commands, i.e., commands that take arguments.

(M (c) (arg_1 ... arg_n) . coms) c an atom. M defines c as a list command. Executing (c e_1 ... e_n) is then performed by substituting e_1 for arg_1, ... e_n for arg_n throughout coms, and then executing coms.

For example, we could define a more general BP by (M (BP) (N) (BK N) UP P). Thus, (BP 3) would perform (BK 3), followed by an UP, followed by a P.

A list command can be defined via a macro so as to take a fixed or indefinite number of 'arguments', as with spread vs. nospread functions. The form given above specified a macro with a fixed number of arguments, as indicated by its argument list. If the 'argument list' is *atomic*, the command takes an indefinite number of arguments.[79]

(M (c) arg . coms) c, arg both atoms, defines c as a list command. Executing (c e_1 ... e_n) is performed by substituting (e_1 ... e_n), i.e., cdr of the command, for arg throughout coms, and then executing coms.

For example, the command 2ND, page 9.30, can be defined as a macro by (M (2ND) X (ORR ((LC . X) (LC . X)))).

---

[79] Note parallelism to EXPR's and EXPR*'s.

Note that for all editor commands, 'built in' commands as well as commands defined by macros, atomic definitions and list definitions are *completely* independent. In other words, the existence of an atomic definition for c in *no* way affects the treatment of c when it appears as car of a list command, and the existence of a list definition for c in *no* way affects the treatment of c when it appears as an atom. In particular, c can be used as the name of either an atomic command, or a list command, or both. In the latter case, two entirely different definitions can be used.

Note also that once c is defined as an atomic command via a macro definition, it will *not* be searched for when used in a location specification, unless it is preceded by an F. Thus (INSERT -- BEFORE BP) would not search for BP, but instead perform a BK, and UP, and a P, and then do the insertion. The corresponding also holds true for list commands.

Occasionally, the user will want to employ the S command in a macro to save some temporary result. For example, the SW command could be defined as:

```
(M (SW) (N M) (NTH N) (S FOO 1) MARK 0 (NTH M) (S FIE 1)
          (I 1 FOO) ←← (I 1 FIE))
```
[80]

Since this version of SW sets foo and fie, using SW may have undesirable side effects, especially when the editor was called from deep in a computation, we would have to be careful to make up unique names for dummy variables used in edit macros, which is bothersome. Furthermore, it would be impossible to define a command that called itself recursively while setting free variables. The BIND command solves both problems.

---

[80] A more elegant definition would be:
(M (SW) (N M) (NTH N) MARK 0 (NTH M) (S FIE 1) (I 1 (## ← 1))
←← (I 1 FIE)), but this would still use one free variable.

(BIND . coms) — binds three dummy variables #1, #2, #3, (initialized to NIL), and then executes the edit commands coms. Note that these bindings are only in effect while the commands are being executed, and that BIND can be used recursively; it will rebind #1, #2, and #3 each time it is invoked.[81]

Thus we could now write SW safely as:

```
(M (SW (N M) (BIND (NTH N) (S #1 1) MARK 0 (NTH M) (S #2 1)
                   (I 1 #1) ←← (I 1 #2)))).
```

User macros are stored on a list usermacros. The prettydef command USERMACROS (Section 14), is available for dumping all or selected user macros.

9.9 Miscellaneous Commands

NIL — unless preceded by F or BF, is always a NOP. Thus extra right parentheses or square brackets at the ends of commands are ignored.

TTY: — calls the editor recursively. The user can then type in commands, and have them executed. The TTY: command is completed when the user exits from the lower editor. (see OK and STOP below).

The TTY: command is extremely useful. It enables the user to set up a complex operation, and perform interactive attention-changing commands part way through

---

[81] BIND is implemented by (PROG (#1 #2 #3) (EDITCOMS (CDR COM))) where com corresponds to the BIND command, and editcoms is an internal editor function which executes a list of commands.

it. For example the command (MOVE 3 TO AFTER COND 3 P TTY:) allows the user to interact, in effect, *within* the MOVE command. Thus he can verify for himself that the correct location has been found, or complete the specification "by hand." In effect, TTY: says "I'll tell you what you should do when you get there."

The TTY: command operates by printing TTY: and then calling the editor. The initial edit chain in the lower editor is the one that existed in the higher editor at the time the TTY: command was entered. Until the user exits from the lower editor, any attention changing commands he executes only affect the lower editor's edit chain.[82] When the TTY: command finishes, the lower editor's edit chain becomes the edit chain of the higher editor.

OK — exits from the editor

STOP — exits from the editor with an error. Mainly for use in conjunction with TTY: commands that the user wants to abort.

Since all of the commands in the editor are errorset protected, the user must exit from the editor via a command.[83] STOP provides a way of distinguishing between a successful and unsuccessful (from the user's standpoint) editing session. For example, if the user is executing (MOVE 3 TO AFTER COND TTY:), and he exits from the lower editor with an OK, the MOVE command will then

---

[82] Of course, if the user performs any structure modification commands while under a TTY: command, these will modify the structure in both editors, since it is the same structure.

[83] Or by typing a control-D. STOP is preferred even if the user is editing at the evalqt level, as it will perform the necessary 'wrapup' to insure that the changes made while editing will be undoable (see Section 22).

complete its operation. If the user wants to abort the MOVE command, he must make the TTY: command generate an error. He does this by exiting from the lower editor with a STOP command. In this case, the higher editor's edit chain will not be changed by the TTY: command.

SAVE — exits from the editor and saves the 'state of the edit' on the property list of the function or variable being edited under the property EDIT-SAVE. If the editor is called again on the same structure, the editing is effectively "continued," i.e., the edit chain, mark list, value of <u>unfind</u> and <u>undolst</u> are restored.

For example:

```
*P
(NULL X)
*F COND P
(COND (& &) (T &))
*SAVE
FOO
   .
   .
   .
←EDITF(FOO)
EDIT
*P
(COND (& &) (T &))
*\ P
(NULL X)
*
```

SAVE is necessary only if the user is editing many different expressions; an exit from the editor via OK always saves the state of the edit of that call to the editor.[84] Whenever the editor is entered, it checks to see if it is editing the same expression as the last one edited. In this case, it restores the mark

---

[84] on the property list of the atom EDIT, under the property name LASTVALUE. OK also remprops EDIT-SAVE from the property list of the function or variable being edited.

list, the undolst, and sets unfind to be the edit chain as of the previous exit from the editor. For example:

```
←EDITF(FOO)
EDIT
*P
(LAMBDA (X) (PROG & & LP & & & &))
     .
     .
     .
*P
(COND & &)
*OK
FOO
     .
     .                                any number of lispx inputs
     .                                except for calls to the editor
←EDITF(FOO)
EDIT
*P
(LAMBDA (X) (PROG & & LP & & & &))
*\ P
(COND & &)
*
```

Furthermore, as a result of the history feature (section 22), if the editor is called on the same expression within a certain number of lispx inputs,[85] the state of the edit of that expression is restored, regardless of how many other expressions may have been edited in the meantime.

---

[85] Namely, the size of the history list, initially 30, but it can be increased by the user.

For example:

```
←EDITF(FOO)
EDIT
*
     .
     .
     .
*P
(COND (& &) (& &) (&) (T &))
*OK
FOO
←    .                less than 30 lispx inputs, including editing
     .
     .
←EDITF(FOO)
EDIT
*\ P
(COND (& &) (& &) (&) (T &))
*
```

Thus the user can always continue editing, including undoing changes from a previous editing session, if

(1) No other expressions have been edited since that session;[86] or

(2) That session was 'sufficiently' recent; or

(3) It was ended with a SAVE command.

* * *

RAISE — is an edit macro defined as UP followed by (I 1 (U-CASE (## 1))), i.e. it raises to upper-case the current expression, or if a tail, the first element of the current expression.

LOWER — Similar to RAISE, except uses l-case.

---

[86] Since saving takes place at *exit* time, intervening calls that were aborted via control-D or exited via STOP will not affect the editor's memory of this last session.

CAP First does a RAISE, and then lowers all but the first character, i.e. the first character is left capitalized.

Note: RAISE, LOWER, and CAP are all NOPs if the corresponding atom or string is already in that state.

(RAISE x) equivalent to (I R (L-CASE x) x), i.e. changes every lower-case x to upper-case in the current expression.

(LOWER x) similar to RAISE, except performs (I R x (L-CASE x)).

Note in both (RAISE x) and (LOWER x), x is typed in in upper case.

REPACK Permits the 'editing' of an atom or string.

For example:

```
*P
... "THIS IS A LOGN STRING")
REPACK
*EDIT
P
(T H I S % I S % A % L O G N % S T R I N G)
*(SW G N)
*OK
"THIS IS A LONG STRING"      87
*
```

REPACK operates by calling the editor recursively on unpack of the current

---

[87] Note that this could also have been accomplished by (R GN NG) or simply (RC GN NG).

expression, or if it is a list, on unpack of its first element. If the lower editor is exited successfully, i.e. via OK as opposed to STOP, the list of atoms is made into a single atom or string, which replaces the atom or string being 'repacked.' The new atom or string is always printed.

(REPACK @) — does (LC . @) followed by REPACK, e.g. (REPACK THIS$).

(; . x) — x is the text of a comment. ; ascends the edit chain looking for a 'safe' place to insert the comment, e.g., in a cond clause, after a prog statement, etc., and inserts (* . x) *after* that point, if possible, otherwise before. For example, if the current expression is (FACT (SUB1 N)) in

```
[COND
    ((ZEROP N) 1)
     (T (ITIMES N (FACT (SUB1 N]
```

(; CALL FACT RECURSIVELY) would insert (* CALL FACT RECURSIVELY) *before* the itimes expression.[88]

; does not change the edit chain, but unfind is set to where the comment was actually inserted.

JOINC — is used to join two neighboring COND's together, e.g. (COND $clause_1$ $clause_2$) followed by

---

[88] If inserted after the itimes, the comment would then be (incorrectly) returned as the value of the cond. However, if the cond was itself a prog statement, and hence its value was not being used, the comment could be (and would be) inserted after the itimes expression.

(COND clause_3 clause_4) becomes (COND clause_1 clause_2 clause_3 clause_4). JOINC does an (F COND T) first so that you don't have to be at the first COND.

(SPLITC x) — splits one COND into two. x specifies the last clause in the first COND, e.g. (SPLITC 3) splits (COND clause_1 clause_2 clause_3 clause_4) into (COND clause_1 clause_2) (COND clause_3 clause_4). Uses generalized NTH command, so that x does not have to be a number, e.g.,the user can say (SPLITC RETURN), meaning split after the clause containing RETURN. SPLITC also does an (F COND T) first.

CL — Clispifies current expression. See Section 23.

DW — Dwimifies current expression. See Section 17 and 23.

(RESETVAR var form . coms) — executes coms while var is reset to the value of form, and then restores var, i.e. effectively calls the function resetvar (Section 5).

9.10 UNDO

Each command that causes structure modification automatically adds an entry to the front of undolst that contains the information required to restore all pointers that were changed by that command.

UNDO — undoes the last, i.e. most recent, structure modification command that has not yet been undone,[89] and prints the name of that command, e.g., MBD UNDONE. The edit chain is then *exactly* what it was before the 'undone' command had been performed.[90] If there are no commands to undo, UNDO types NOTHING SAVED.

!UNDO — undoes all modifications performed during this editing session, i.e. this call to the editor. As each command is undone, its name is printed a la UNDO. If there is nothing to be undone, !UNDO prints NOTHING SAVED.

---

[89] Since UNDO and !UNDO cause structure modification, they also add an entry to undolst. However, UNDO and !UNDO entries are skipped by UNDO, e.g., if the user performs an INSERT, and then an MBD, the first UNDO will undo the MBD, and the second will undo the INSERT. However, the user can also specify precisely which commands he wants undone by identifying the corresponding entry on the history list as described in Section 22. In this case, he can undo an UNDO command, e.g. by typing UNDO UNDO, or undo a !UNDO command, or undo a command other than that most recently performed.

[90] Undoing an event containing an I, E, or S command will also undo the side effects of the evaluation(s), e.g. undoing (I 3 (/NCONC FOO FIE)) will not only restore the 3rd element but also restore FOO. Similarly, undoing an S command will undo the set. See discussion of UNDO in Section 22. (Note that if the I command was typed directly to the editor, /NCONC would automatically be substituted for NCONC as described in Section 22.)

Whenever the user *continues* an editing session as described on page 9.72-74, the undo information of the previous session is protected by inserting a special blip, called an undo-block, on the front of <u>undolst</u>. This undo-block will terminate the operation of a !UNDO, thereby confining its effect to the current session, and will similarly prevent an UNDO command from operating on commands executed in the previous session.

Thus, if the user enters the editor continuing a session, and immediately executes an UNDO or !UNDO, the editor will type BLOCKED instead of NOTHING SAVED. Similarly, if the user executes several commands and then undoes them all, another UNDO or !UNDO will also cause BLOCKED to be typed.

UNBLOCK — removes an undo-block. If executed at a non-blocked state, i.e. if UNDO or !UNDO *could* operate, types NOT BLOCKED.

TEST — adds an undo-block at the front of <u>undolst</u>.

Note that TEST together with !UNDO provide a 'tentative' mode for editing, i.e. the user can perform a number of changes, and then undo all of them with a single !UNDO command.

9.11 Editdefault

Whenever a command is not recognized, i.e., is not 'built in' or defined as a macro, the editor calls an internal function, editdefault, to determine what action to take.[91] If a location specification is being executed, an internal flag informs editdefault to treat the command as though it had been preceded by an F.

If the command is a list, an attempt is made to perform spelling correction on car of the command[92] using editcomsl, a list of all list edit commands.[93] If spelling correction is successful,[94] the correct command name is rplacaed into the command, and the editor continues by executing the command. In other words, if the user types (LP F PRINT (MBBD AND (NULL FLG))), only one spelling correction will be necessary to change MBBD to MBD. If spelling correction is not successful, an error is generated.

If the command is atomic, the procedure followed is a little more elaborate.

---

[91] Since editdefault is part of the edit block, the user cannot advise or redefine it as a means of augmenting or extending the editor. However, the user can accomplish this via edituserfn. If the value of the variable edituserfn is T, editdefault calls the function edituserfn giving it the command as an argument. If edituserfn returns a non-NIL value, its value is interpreted as a single command and executed. Otherwise, the error correction procedure described below is performed.

[92] unless dwimflg=NIL. See Section 17 for discussion of spelling correction.

[93] When a macro is defined via the M command, the command name is added to editcomsa or editcomsl, depending on whether it is an atomic or list command. The prettydef command USERMACROS (Section 14), is aware of this, and provides for restoring editcomsa and editcomsl.

[94] Throughout this discussion, if the command was not typed in directly, the user will be asked to approve the spelling correction. See Section 17.

1) If the command is one of the list commands, i.e., a member of <u>editcomsl</u>, and there is additional input on the same teletype line, treat the entire line as a single list command.[95] Thus, the user may omit parentheses for any list command typed in at the top level (provided the command is not also an atomic command, e.g. NX, BK). For example,

```
*P
(COND (& &) (T &))
*XTR 3 2]
*MOVE TO AFTER LP
*
```

If the command is on the list <u>editcomsl</u> but no additional input is on the teletype line, an error is generated, e.g.

```
*P
(COND (& &) (T &))
*MOVE

MOVE?
*
```

If the command is on <u>editcomsl</u>, and *not* typed in directly, e.g. it appears as one of the commands in a LP command, the procedure is similar, with the rest of the command stream at that level being treated as "the teletype line", e.g.

(LP F (COND (T &)) XTR 2 2).[96]

2) If the command was typed in and the first character in the command is an 8,

---

[95] The line is read using <u>readline</u> (Section 14). Thus the line can be terminated by a square bracket, or by a carriage return not preceded by a space.

[96] Note that if the command is being executed in location context, <u>editdefault</u> does not get this far, e.g. (MOVE TO AFTER COND XTR 3) will search for XTR, *not execute it*. However, (MOVE TO AFTER COND (XTR 3)) will work.

treat the 8 as a mistyped left parenthesis, and and the rest of the line as the arguments to the command, e.g.,

```
*P
(COND (& &) (T &))
*8-2 (Y (RETURN Z)))
=(-2
*P
(COND (Y &) (& &) (T &))
```

3) If the command was typed in, is the name of a function, and is followed by NIL or a list car of which is not an edit command, assume the user forgot to type E and means to apply the function to its arguments, type =E and the function name, and perform the indicated computation, e.g.

```
*BREAK(FOO)
=E BREAK
(FOO)
*
```

4) If the last character in the command is P, and the first n-1 characters comprise a number, assume that the user intended two commands, e.g.,

```
*P
(COND (& &) (T &))
*0P
=0 P
(SETQ X (COND & &))
```

5) Attempt spelling correction using editcomsa, and if successful,[97] execute the corrected command.

6) Otherwise, if there is additional input on the same line, or command stream, spelling correct using editcomsl, e.g.,

---

[97] See footnote on page 9.81.

```
*MBBD SETQ X
=MBD
*
```

7) Otherwise, generate an error.

9.12 Editor Functions

edite[expr;coms;atm] — edits an expression. Its value is the last element of editl[list[expr];coms;atm]. Generates an error if expr is not a list.

editl[l;coms;atm;mess] — editl[98] *is* the editor. Its first argument is the edit chain, and its value is an edit chain, namely the value of l at the time editl is exited.[99]

coms is an optional list of commands. For interactive editing, coms is NIL. In this case, editl types EDIT and then waits for input from teletype.[100] Exit occurs only via an OK, STOP, or SAVE command.

If coms is *not* NIL, no message is typed, and each

---

[98] edit-*ell*, not edit-one.

[99] l is a specvar, and so can be examined or set by edit commands. For example, ↑ is equivalent to (E (SETQ L (LAST L)) T). However, the user should only manipulate or examine l directly as a last resort, and then with caution.

[100] If mess is not NIL, editl types it instead of EDIT. For example, the TTY: command is essentially (SETQ L (EDITL L NIL NIL (QUOTE TTY:))).

member of coms is treated as a command and executed. If an error occurs in the execution of one of the commands, no error message is printed, the rest of the commands are ignored, and editl exits with an error, i.e. the effect is the same as though a STOP command had been executed. If all commands execute successfully, editl returns the current value of l.

atm is optional. On calls from editf, it is the name of the function being edited; on calls from editv, the name of the variable, and calls from editp, the atom whose property list is being edited. The property list of atm is used by the SAVE command for saving the state of the edit. Thus SAVE will not save anything if atm=NIL, i.e. when editing arbitrary expressions via edite or editl directly.

editl0[l;coms;mess;editlflg][101] like editl except does not rebind or initialize the editor's various state variables, such as lastail, unfind, undolst, marklst, etc.

editf[x] nlambda, nospread function for editing a function. car[x] is the name of the function,cdr[x] an optional list of commands. For the rest of the discussion, fn is car[x], and coms is cdr[x].

The value of editf is fn.

---

[101] editlflg=T is for internal use by the editor.

(1) In the most common case, fn is an expr, and editf simply performs putd[fn;edite[getd[fn];coms;fn]].

(2) If fn is not an expr, but has an EXPR property, editf prints PROP, and performs edite[getp[fn;EXPR];coms;fn]. If edite returns (i.e. if the editing is not terminated by a STOP), and some changes were made, editf performs unsavedef[fn], prints UNSAVED, and then does putd[fn;value-of-edite].

(3) If fn is neither an expr nor has an EXPR property, but its top level value is a list, editf assumes the user meant to call editv, prints =EDITV, calls editv and returns. Similarly, if fn has a non-NIL property list, editf prints =EDITP, calls editp and returns.

(4) If fn is neither a function, nor has an EXPR property, nor a top level value that is a list, nor a non-NIL property list, editf attempts spelling correction using the spelling list userwords,[102] and if successful, goes back to (1).

Otherwise, editf generates an fn NOT EDITABLE error.

If editf ultimately succeeds in finding a function to edit, i.e. does not exit by calling editv or editp, editf calls the function addspell after editing has been completed.[103] Addspell 'notices' fn, i.e. sets lastword to fn, and adds fn

---

[102] Unless dwimflg=NIL. Spelling correction is performed using the function misspelled?. If fn=NIL, misspelled? returns the last 'word' referenced, e.g. by defineq, editf, prettyprint etc. Thus if the user defines foo and then types editf[], the editor will assume he meant foo, type =FOO, and then type EDIT. See Section 17.

[103] Unless dwimflg=NIL. addspell is described in Section 17.

to the appropriate spelling lists. If any changes were made, editf also calls newfile?, which performs the updating for the file package as described in Section 14.

editv[editvx] — nlambda, nospread function, similar to editf, for editing values. car[editvx] specifies the value, cdr[editvx] is an optional list of commands.

If car[editvx] is a list, it is evaluated and its value given to edite, e.g. EDITV((CDR (ASSOC (QUOTE FOO) DICTIONARY))). In this case, the value of editv is T.

However, for most applications, car[editvx] is a variable name, i.e. atomic, as in EDITV(FOO). If the value of this variable is NOBIND, editv checks to see if it is the name of a function, and if so, assumes the user meant to call editf, prints =EDITF, calls editf and returns. Otherwise, editv attempts spelling correction using the list userwords.[104] Then editv will call edite on the value of car[editvx] (or the corrected spelling thereof). Thus, if the value of foo is NIL, and the user performs (EDITV FOO), no spelling correction will occur, since foo is the name of a variable in the user's system, i.e. it has a value. However, edite will generate an error, since foo's value is not a list, and hence not editable. If the user performs (EDITV FOOO), where the value of fooo is NOBIND, and foo is on the user's spelling list, the spelling corrector will correct FOOO to FOO. Then edite will be called on the value of foo. Note that this may still result in an error if the value of foo is not a list.

When (if) edite returns, editv sets the variable to the value returned, and calls addspell and newfile?.

---

[104] Unless dwimflg=NIL. Misspelled? is also called if car[editvx] is NIL, so that EDITV() will edit lastword.

The value of editv is the name of the variable whose value was edited.

editp[x] nlambda, nospread function, similar to editf for editing property lists. If the property list of car[x] is NIL, editp attempts spelling correction using userwords. Then editp calls edite on the property list of car[x], (or the corrected spelling thereof). When (if) edite returns, editp rplacd's car[x] with the value returned, and calls addspell.

The value of editp is the atom whose property list was edited.

editfns[x] nlambda, nospread function, used to perform the same editing operations on several functions. car[x] is evaluated to obtain a list of functions. cdr[x] is a list of edit commands. editfns maps down the list of functions, prints the name of each function, and calls the editor (via editf) on that function.[105]

For example, EDITFNS(FOOFNS (R FIE FUM)) will change every FIE to FUM in each of the functions on foofns.

The call to the editor is errorset protected, so

---

[105] i.e. the definition of editfns might be:

```
[MAPC (EVAL (CAR X)) (FUNCTION (LAMBDA (Y)
        (APPLY (QUOTE EDITF)
               (CONS (PRINT Y T) (CDR X]
```

that if the editing of one function causes an error, editfns will proceed to the next function.[106]

Thus in the above example, if one of the functions did not contain a FIE, the R command would cause an error, but editing would continue with the next function.

The value of editfns is NIL.

edit4e[pat;x;changeflg] is the pattern match routine. Its value is T if pat-matches x. See page 9.21-23 for definition of 'match'.[107]

Note: before each search operation in the editor begins, the entire pattern is scanned for atoms or strings containing alt-modes. These are replaced by patterns of the form (CONS (QUOTE $) (UNPACK atom/string)) for 6a, and (CONS (QUOTE $$) (CONS (NCHARS atom/string) (UNPACK atom/string))), for 6b.[108] Thus from the standpoint of edit4e, pattern type 6a is indicated by car[pat] being the atom $ ($ is alt-mode) and pattern type 6b by car[pat] being the atom $$ (double alt-mode).

---

[106] In particular, if an error occurred while editing a function via its EXPR property, the function would not be unsaved. In other words, in the above example, only those functions which contained a FIE, i.e. only those actually changed, would be unsaved.

[107] changeflg is for internal use by the editor.

[108] In latter case, atom/string corresponds to the atom or string up to but not including the final two-alt-modes. In both cases, dunpack is used wherever possible.

Therefore, if the user wishes to call edit4e directly, he must first convert any patterns which contain atoms or strings ending in alt-modes to the form recognized by edit4e. This is done with the function editfpat.

editfpat[pat;flg] makes a copy of pat with all patterns of type 6 converted to the form expected by edit4e.[109]

editfindp[x;pat;flg] allows a program to use the edit find command as a pure predicate from outside the editor. x is an expression, pat a pattern. The value of editfindp is T if the command F pat *would* succeed, NIL otherwise. editfindp calls editfpat to convert pat to the form expected by edit4e, unless flg=T. Thus, if the program is applying editfindp to several different expressions using the same pattern, it will be more efficient to call editfpat once, and then call editfindp with the converted pattern and flg=T.

esubst[x;y;z;errorflg;charflg] equivalent to performing (R y x)[110] with z as the current expression, i.e. the order of arguments is the same as for subst. Note that y and/or x can employ alt-modes. The value of esubst is the modified z. Generates an error[111]

---

[109] flg=T is used for internal use by the editor.

[110] unless charflg=T, in which case it is equivalent to (RC y x). See page 9.59.

[111] of the type that never causes a break.

if y not found in z. If errorflg=T, also prints an error message of the form y ?.

esubst is always undoable.

changename[fn;from;to] — replaces all occurrences of from by to in the definition of fn. If fn is an expr, changename performs nlsetq[esubst[to;from;getd[fn]]]. If fn is *compiled*, changename searches the literals of fn (and all of its compiler generated subfunctions), replacing each occurrence of from with to.[112]

The value of changename is fn if at least one instance of from was found, otherwise NIL.

changename is used by break and advise for changing calls to fn1 to calls to fn1-IN-fn2.

editracefn[com] — is available to help the user debug complex edit macros, or subroutine calls to the editor. If editracefn is set to T, the function editracefn is called whenever a command that was not typed in by the user is about to be executed, giving it that command as its argument. However, the TRACE and BREAK options described below are probably sufficient for most applications.

---

[112] Will succeed even if from is called from fn via a linked call. In this case, the call will also be relinked to call to instead.

If editracefn is set to TRACE, the name of the command and the current expression are printed. If editracefn=BREAK, the same information is printed, and the editor goes into a break. The user can then examine the state of the editor.

editracefn is initially NIL.

Page Numbers

	Page Numbers

	Page Numbers
(; . x) (edit command)	9.76
= (typed by editor)	9.12
== (in edit pattern)	9.22
=E (typed by editor)	9.82
=EDITF (typed by editor)	9.86
=EDITP (typed by editor)	9.85
=EDITV (typed by editor)	9.85
? (edit command)	9.2,60
? (typed by editor)	9.3
@ (location specification) (in editor)	9.29
(@1 THRU @2) (edit command)	9.54
(@1 THRU) (edit command)	9.56
(@1 TO @2) (edit command)	9.54
(@1 TO) (edit command)	9.56
\ (edit command)	9.11,34-35,41
(\ atom) (edit command)	9.34
\P (edit command)	9.11,35,61
↑ (edit command)	9.4,18
← (edit command)	9.34
(← pattern) (edit command)	9.30
←← (edit command)	9.34

SECTION 10

ATOM, STRING, ARRAY, AND STORAGE MANIPULATION

10.1 Pnames and Atom Manipulation

The term 'print name' (of an atom) in LISP 1.5 referred to the characters that were output whenever the atom was printed. Since these characters were stored on the atom's property list under the property PNAME, pname was used interchangeably with 'print name'. In INTERLISP, all pointers have pnames, although only literal atoms and strings have their pname explicitly stored.

The pname of a pointer are those characters that are output when the pointer is printed using prin1,

e.g., the pname of the atom ABC%(D[1] consists of the five characters ABC(D. The pname of the list (A B C) consists of the seven characters (A B C) (two of the characters are spaces).

Sometimes we will have occasion to refer to the prin2-pname.

The prin2-pname are those characters output when the corresponding pointer is printed using prin2.

Thus the prin2-pname of the atom ABC%(D is the *six* characters ABC%(D. Note that the pname of numbers depends on the setting of radix.

---

[1] % is the escape character. See Sections 2 and 14.

pack[x]

If x is a list of atoms, the value of pack is a single atom whose pname is the concatenation of the pnames of the atoms in x, e.g.

pack[(A BC DEF G)]=ABCDEFG.

If the pname of the value of pack[x] is the same as that of a number, pack[x] will be that number, e.g. pack[(1 3.4)]=13.4,

pack[(1 E -2)]=.01.

Although x is usually a list of atoms, it can be a list of arbitrary INTERLISP pointers. The value of pack is still a single atom whose pname is the same as the concatenation of the pnames of all the pointers in x, e.g.

pack[((A B)"CD")] = %(A% B%)CD.

In other words, mapc[x;prin1] and prin1[pack[x]] always produce *exactly* the same output. In fact, pack actually operates by calling prin1 to convert the pointers to a stream of characters (without printing) and then makes an atom out of the result.

Note: atoms are restricted to < 99 characters. Attempting to create a larger atom either via pack or by typing one in (or reading from a file) will cause an error, ATOM TOO LONG.

unpack[x;flg]

The value of unpack is the pname of x as a list of characters (atoms),[2] e.g.

---

[2] There are no special 'character-atoms' in INTERLISP, i.e. an atom consisting of a single character is the same as any other atom.

unpack[ABC] = (A B C)

unpack["ABC(D"] = (A B C %(D)

In other words prin1[x] and mapc[unpack[x];prin1] produce the same output.

If flg=T, the prin2-pname of x is used, e.g. unpack["ABC(D";T]= (%" A B C %(D %").

Note: unpack[x] performs n conses, where n is the number of characters in the pname of x.

dunpack[x;scratchlist;flg] — a destructive version of unpack that does not perform any conses but instead uses scratchlist to make a list equal to unpack[x;flg]. If the p-name is too long to fit in scratchlist, dunpack calls unpack and returns unpack[x;flg]. Gives an error if scratchlist is not a list.

nchars[x;flg] — number of characters in pname of x.[3] If flg=T, the prin2-pname is used. E.g. nchars["ABC"]=3, nchars["ABC";T]=5.

nthchar[x;n;flg] — Value is nth character of pname of x. Equivalent to car[nth[unpack[x;flg];n]] but faster and does no conses. n can be negative, in which case counts from end of pname, e.g. -1 refers to the

---

[3] Both nthchar and nchars work much faster on objects that actually have an internal representation of their pname, i.e. literal atoms and strings, than they do on numbers and lists, as they do not have to simulate printing.

last character, -2 next to last, etc. If n is greater than the number of characters in the pname, or less than minus that number, or 0, the value of nthchar is NIL.

packc[x] — like pack except x is a list of (ASCII) character codes, e.g. packc[(70 79 79)]=FOO.

chcon[x;flg] — like unpack, except returns the pname of x as a list of (ASCII) character codes, e.g. chcon[FOO] = (70 79 79). If flg=T, the prin2-pname is used.

chcon1[x] — returns character code of first character of pname of x, e.g. chcon1[FOO] = 70. Thus chcon[x] could be written as mapcar[unpack[x];chcon1].

dchcon[x;scratchlist;flg] — similar to dunpack

character[n] — n is an ASCII character *code*. Value is the atom having the corresponding single character as its pname,[4] e.g. character[70] = F. Thus, unpack[x] could be written as mapcar[chcon[x];character].

fcharacter[n] — fast version of character that compiles open.

gensym[char] — Generates a new atom of the form xnnnn, where x=char (or A if char is NIL) in which each of the

---

[4] See footnote 2.

n's is a digit. Thus, the first one generated is A0001, the second A0002, etc. gensym provides a way of generating new atoms for various uses within the system. The value of gennum, initially 10000, determines the next gensym, e.g. if gennum is set to 10023, gensym[ ]=A0024.

The term gensym is used to indicate an atom that was produced by the function gensym. Atoms generated by gensym are the same as any other literal atoms: they have property lists, and can be given function definitions. Note that the atoms are not guaranteed to be new.

For example, if the user has previously created A0012, either by typing it in, or via pack or gensym itself, when gennum gets to 10011, the next value returned by gensym will be the A0012 already in existence.

mapatoms[fn] — Applies fn to every literal atom in the system, e.g. mapatoms[(LAMBDA(X)(AND(SUBRP X)(PRINT X)))] will print every subr. Value of mapatoms is NIL.

10.2 String Functions

stringp[x] — Is x if x a string, NIL otherwise. Note: if x is a string, nlistp[x] is T, but atom[x] is NIL.

strequal[x;y] — Is x if x and y are both strings and equal, i.e. print the same, otherwise NIL. Equal uses strequal. Note that strings may be equal without being eq.

mkstring[x] — Value is string corresponding to prin1 of x.

rstring[ ] — Reads a string - see Section 14.

substring[x;n;m]

Value is the substring of x consisting of the nth thru mth characters of x. If m is NIL, the substring is the nth character of x thru the end of x. n and m can be negative numbers, as with nthchar. Returns NIL if the substring is not well defined, e.g. n or m > nchars[x] or < minus[nchars[x]] or n corresponds to a character in x to the right of the character indicated by m.

If x is not a string, equivalent to substring[mkstring[x];n;m], except substring does not have to actually make the string if x is a literal atom.[5] For example, substring[(A B C);4;6]="B C".

gnc[x]

get next character of string x. Returns the next character of the string, (as an atom), *and* removes the character from the string. Returns NIL if x is the null string. If x isn't a string, a string is made. Used for sequential access to characters of a string.

Note that if x is a substring of y, gnc[x] does not remove the character from y, i.e. gnc doesn't physically change the string of characters, just the pointer and the byte count.[6]

---

5 See string storage section that follows.

6 See string storage section that follows.

glc[x] — gets last character of string x. Above remarks about gnc also supply to glc.

$concat[x_1;x_2;...;x_n]$ — lambda nospread function. Concatenates (copies of) any number of strings. The arguments are transformed to strings if they aren't strings. Value is the new string, e.g. concat["ABC";DEF;"GHI"] = "ABCDEFGHI". The value of concat[] is the null string, "".

rplstring[x;n;y] — Replace characters of string x beginning at character n with string y. n may be positive or negative. x and y are converted to strings if they aren't already. Characters are smashed into (converted) x. Returns new x. Error if there is not enough room in x for y, i.e. the new string would be longer than the original.[7] Note that if x is a substring of z, z will also be modified by the action of rplstring.

mkatom[x] — Creates an atom whose pname is the same as that of the string x or if x isn't a string, the same as that of mkstring[x], e.g. mkatom[(A B C)] is the atom %(A% B% C%). If atom would have > 99 characters, causes an error, ATOM TOO LONG.

---

[7] If y was not a string, x will already have been partially modified since rplstring does not know whether y will 'fit' without actually attempting the transfer.

Searching Strings

strpos is a function for searching one string looking for another. Roughly it corresponds to member, except that it returns a character position number instead of a tail. This number can then be given to substring or utilized in other calls to strpos.

strpos[x;y;start;skip;anchor;tail]

x and y are both strings (or else they are converted automatically). Searches y beginning at character number start, (or else 1 if start is NIL) and looks for a sequence of characters equal to x. If a match is found, the corresponding character position is returned, otherwise NIL, e.g.,

strpos["ABC","XYZABCDEF"]=4

strpos["ABC","XYZABCDEF";5]=NIL

strpos["ABC","XYZABCDEFABC";5]=10

skip can be used to specify a character in x that matches any character in y, e.g.

strpos["A&C&";"XYZABCDEF";NIL;&]=4

If anchor is T, strpos compares x with the characters beginning at position start, or 1. If that comparison fails, strpos returns NIL without searching any further down y. Thus it can be used to compare one string with some *portion* of another string, e.g.

strpos["ABC";"XYZABCDEF";NIL;NIL;T]=NIL

strpos["ABC";"XYZABCDEF";4;NIL;T]=4

Finally, if tail is T, the value returned by strpos if successful is not the starting position of the sequence of characters corresponding to x, but the position of the first character after that, i.e. starting point plus nchars[x] e.g. strpos["ABC";"XYZABCDEFABC";NIL;NIL;NIL;T]=7. Note that strpos["A";"A";NIL;NIL;NIL;T]=2, even though "A" has only one character.

Example Problem

Given the strings x, y, and z, write a function foo that will make a string corresponding to that portion of x between y and z, e.g. foo["NOW IS THE TIME FOR ALL GOOD MEN";"IS";"FOR"] is " THE TIME ".

Solution:

```
(FOO
  [LAMBDA (X Y Z)
    (AND (SETQ Y (STRPOS Y X NIL NIL NIL T))
         (SETQ Z (STRPOS Z X Y))
         (SUBSTRING X Y (SUB1 Z])
```

strposl[a;str;start;neg] — str is a string (or else it is converted automatically to a string), a is a list of characters or character codes.[8] strposl searches str beginning at character number start (or else 1 if start=NIL) for one of the characters in a. If one is found, strposl returns as its value the

---

[8] If any element of a is a number, it is assumed to be a character code. Otherwise, it is converted to a character code via chcon1. Therefore, it is more efficient to call strposl with a a list of character *codes*.

corresponding character position, otherwise NIL. E.g., strposl[(A B C);"XYZBCD"]=4. If neg=T, strposl searches for a character *not* on a, e.g., strposl[(A B C); "ABCDEF";NIL;T]=4.

If a is an array, it is treated as a bit table. The bits of (ELT A 1) correspond to character codes 0 to 43Q, of (ELT A 2) to codes 44Q to 107Q, etc. Thus an array whose first element was 17Q would be equivalent to a list (40Q 41Q 42Q 43Q) or (%_ ! %" #).

If a is not a bit table (array), strposl first converts it to a bit table using makebittable described below. If strposl is to be called frequently with the same list of characters, a considerable savings can be achieved by converting the list to a bit table *once*, and then passing the bit table to strposl as its first argument.

makebittable[l;neg;a] makes a bit table suitable for use by strposl. l and neg are as for strposl. If a is not an array with at least 4 elements, makebittable will create an array and return that as its value. Otherwise it uses (and changes) a.

Note: if neg=T, strposl must call makebittable whether a is a list *or* an array. To obtain bit table efficiency with neg=T, makebittable should be called with neg=T, to construct the "inverted" table, and the resulting table (array) should be given to strposl with neg=NIL.

String Storage

A string is stored in 2 parts; the characters of the string, and a pointer to the characters. The pointer, or 'string pointer', indicates the *byte* at which the string begins and the length of the string. It occupies one word of storage. The characters of the string are stored five characters to a word in a portion of the INTERLISP address space devoted exclusively to storing characters.

Since the internal pname of literal atoms also consists of a pointer to the beginning of a string of characters and a byte count, conversion between literal atoms and strings does not require any additional storage for the characters of the pname, although one cell is required for the string pointer.[9]

When the conversion is done internally, e.g. as in substring, strpos, or strposl, no additional storage is required for using literal atoms instead of strings.

The use of storage by the basic string functions is given below:

mkstring[x]	x string	no space
	x literal atom	new pointer
	other	new characters and pointer
substring[x;n;m]	x string	new pointer
	x literal atom	new pointer
	other	new characters and pointer

---

[9] Except when the string is to be smashed by rplstring. In this case, its characters must be copied to avoid smashing the pname of the atom. rplstring automatically performs this operation.

gnc[x] and glc[x]	x string	no space, pointer is modified
	other	like mkstring, but doesn't make much sense
concat[x_1;x_2;...x_n]	args any type	new characters for whole new string, one new pointer
rplstring[x;n;y]	x string	no new space unless characters are in pname space (as result of mkstring[atom]) in which case x is quietly copied to string space
	x other	new pointer and characters
	y any type	type of y doesn't matter

10.3 Array Functions

Space for arrays and compiled code are both allocated out of a common array space. Arrays of pointers and unboxed numbers may be manipulated by the following functions:

array[n;p;v] This function allocates a block of n+2 words, of which the first two are header information. The next $p \leq n$ are cells which will contain unboxed numbers, and are initialized to unboxed 0. The last $n-p \geq 0$ cells will contain pointers initialized with v, i.e., both car and cdr are available for storing information, and each initially contain v. If p is NIL, 0 is used (i.e., an array containing all INTERLISP pointers). The value of array is the array, also

called an array pointer. If sufficient space is not available for the array, a garbage collection of array space, GC: 1, is initiated. If this is unsuccessful in obtaining sufficient space, an error is generated, ARRAYS FULL.

Array-pointers print as #n, where n is the octal representation of the pointer. Note that #n will be read as a literal atom, and not an array pointer.

arraysize[a] — Returns the size of array a. Generates an error, ARG NOT ARRAY, if a is not an array.

arrayp[x] — Value is x if x is an array pointer otherwise NIL. No check is made to ensure that x actually addresses the beginning of an array.

elt[a;n] — Value is nth element of the array a[10] elt generates an error, ARG NOT ARRAY, if a is not the beginning of an array.[11] If n corresponds to the *unboxed* region of a, the value of elt is the full 36 bit word, as a boxed integer. If n corresponds to the *pointer* region of a, the value of elt is the car half of the corresponding element.

seta[a;n;v] — sets the nth element of the array a. Generates an

---

[10] elt[a;1] is the first element of the array (actually corresponds to the 3rd cell because of the 2 word header).

[11] arrayp is true for pointers into the middle of arrays, but elt and seta must be given a pointer to the beginning of an array, i.e., a value of array.

error, ARG NOT ARRAY, if a is not the beginning of an array. If n corresponds to the *unboxed* region of a, v must be a number, and is unboxed and stored as a full 36 bit word into the nth element of a. If n corresponds to the *pointer* region of a, v replaces the car half of the nth element. The value of seta is v.

Note that seta and elt are always inverse operations.

eltd[a;n] — same as elt for unboxed region of a, but returns cdr half of nth element, if n corresponds to the pointer region of a.

setd[a;n;v] — same as seta for unboxed region of a, but sets cdr half of nth element, if n corresponds to the pointer region of a. The value of setd is v.

In other words, eltd and setd are always inverse operations.

10.4 Storage Functions

reclaim[n] — Initiates a garbage collection of type n. Value of reclaim is number of words available (for that type) after the collection.

Garbage collections, whether invoked directly by the user or indirectly by need for storage, do not confine their activity solely to the data type for which they were called, but automatically collect some or all of the other types (see Section 3).

ntyp[x] — Value is type number for the data type of INTERLISP pointer x, e.g. ntyp[(A . B)] is 8, the type number for lists. Thus GC: 8 indicates a garbage collection of list words.

type	number
arrays, compiled code	1
stack positions	2
list words	8
atoms	12
floating point numbers	16
large integers	18
small integers	20
string pointers	24
pname storage	28
string storage	30

typep[x;n]

eq[ntyp[x];n]

gcgag[message]

message is a string or atom to be printed (using prin1) wherever a garbage collection is begun. If message=T, its standard setting, GC: is printed, followed by the type number. When the garbage collection is complete, two numbers are printed the number of words collected for that type, and the total number of words available for that type, i.e. allocated but not necessarily currently in use (see minfs below).

Example:

```
←RECLAIM(18)

GC: 18
511, 3071 FREE WORDS
3071
←RECLAIM(12)

GC: 12
1020, 1020 FREE WORDS
1020
```

If message=NIL, no garbage collection message is printed, either on entering or leaving the garbage collector. Value of gcgag is old setting.

minfs[n;typ]

Sets the minimum amount of free storage which will be maintained by the garbage collector for data types of type number typ. If, after any garbage collection for that type, fewer than n free words are present, sufficient storage will be added (in 512 word chunks) to raise the level to n.

If typ=NIL, 8 is used, i.e. the minfs refers to list words.

If n=NIL, minfs returns the current minfs setting for the corresponding type.

A minfs setting can also be changed dynamically, even during a garbage collection, by typing control-S followed by a number, followed by a period.[12] If the control-S was typed during a garbage collection, the number is the new minfs setting for the type being collected, otherwise for type 8, i.e. list words.

Note: A garbage collection of a 'related' type may also cause more storage to be assigned to that type. See discussion of garbage collector algorithm, Section 3.

storage[flg]

Prints amount of storage (by type number) used by and assigned to the user, e.g.

---

12 When the control-S is typed, INTERLISP immediately clears and saves the input buffer, rings the bell, and waits for input, which is terminated by any non-number. The input buffer is then restored, and the program continues. If the input was terminated by other than a period, it is ignored.

←STORAGE()

TYPE	USED	ASSIGNED
1	80072	87552
8	7970	9216
12	7032	7680
16	0	512
18	1124	2560
24	118	512
28	4226	4608
30	573	1024
SUM	101115	113664

If flg=T, includes storage used by and assigned to the system. Value is NIL.

gctrp[n]

garbage collection trap. Causes a (simulated) control-H interrupt when the number of free list words (type 8) remaining equals n, i.e. when a garbage collection would occur in n more conses. The message GCTRP is printed, the function interrupt (Section 16) is called, and a break occurs. Note that by advising (Section 19) interrupt the user can program the handling of a gctrp instead of going into a break.[13]

Value of gctrp is its last setting.

gctrp[-1] will 'disable' a previous gctrp since there are never -1 free list words. gctrp is initialized this way.

---

[13] For gctrp interrupts, interrupt is called with intype (its third argument) equal to 3. If the user does not want to go into a break, the advice should still allow interrupt to be entered, but first set intype to -1. This will cause interrupt to "quietly" go away by calling the function that was interrupted. The advice should *not* exit interrupt via return, as in this case the function that was about to be called when the interrupt occurred would not be called.

gctrp[] returns number of list words left, i.e. number of conses until next type 8 garbage collection, see Section 21.

conscount[n]

conscount[] returns number of conses since INTERLISP started up. If n is not NIL, resets conscount to n.

closer[a;x]

Stores x into memory location a. Both x and a must be numbers.

openr[a]

Value is the number in memory location a, i.e. boxed.

Index for Section 10

Page Numbers

SECTION 11

FUNCTIONS WITH FUNCTIONAL ARGUMENTS

As in all LISP 1.5 Systems, arguments can be passed which can then be used as functions. However, since car of a form is *never* evaluated, apply or apply* must be used to call the function specified by the value of the functional argument.

Functions which use functional arguments should use variables with obscure names to avoid possible conflict with variables that are used by the functional argument. For example, all system functions standardly use variable names consisting of the function name concatenated with x or fn, e.g. mapx. Note that by specifying the free variables used in a functional argument as the second argument to function, thereby using the INTERLISP FUNARG feature, the user can be sure of no clash.

function[x;y] — is an nlambda function. If y=NIL, the value of function is identical to quote, for example, (MAPC LST (FUNCTION PRINT)) will cause mapc to be called with two arguments the value of lst and PRINT. Similarly, (MAPCAR LST (FUNCTION(LAMBDA(Z) (LIST (CAR Z))))) will cause mapcar to be called with the value of lst and (LAMBDA (Z) (LIST (CAR Z))). When compiled, function will cause code to be compiled for x; quote will not. Thus

(MAPCAR LST (QUOTE (LAMBDA --))) will cause mapcar to be called with the value of lst and the expression (LAMBDA --). The functional argument will therefore still be interpreted. The corresponding expression using function will cause a dummy function to be created with definition (LAMBDA --), and then compiled. mapcar would then be called with the value of lst and the name of the dummy function. See Section 18.

If y is not NIL, it is a list of variables that are (presumably) used freely by x. In this case, the value of function is an expression of the form (FUNARG x array), where array contains the variable bindings for those variables on y. Funarg is described on page 11.5-7.

map[mapx;mapfn1;mapfn2]

If mapfn2 is NIL, map applies the function mapfn1 to successive tails of the list mapx. That is, first it computes mapfn1[mapx], and then mapfn1[cdr[mapx]], etc., until mapx is exhausted.[1] If mapfn2 is provided, mapfn2[mapx] is used instead of cdr[mapx] for the next call for mapfn1, e.g., if mapfn2 were cddr, alternate elements of the list would be skipped.

The value of map is NIL. map compiles open.

---

[1] i.e., becomes a non-list.

mapc[mapx;mapfn1;mapfn2] Identical to _map_, except that mapfn1[car[mapx]] is computed at each iteration instead of mapfn1[mapx], i.e., _mapc_ works on elements, _map_ on tails. The value of _mapc_ is NIL. _mapc_ compiles open.

maplist[mapx;mapfn1;mapfn2] successively computes the same values that _map_ would compute; and returns a list consisting of those values. _maplist_ compiles open.

mapcar[mapx;mapfn1;mapfn2] computes the same values that _mapc_ would compute, and returns a list consisting of those values, e.g. mapcar[x;FNTYP] is a list of _fntyps_ for each element on _x_. _mapcar_ compiles open.

mapcon[mapx;mapfn1;mapfn2] Computes the same values as _map_ and _maplist_ but _nconcs_ these values to form a list which it returns. _mapcon_ compiles open.

mapconc[mapx;mapfn1;mapfn2] Computes the same values as _mapc_ and _mapcar_, but _nconcs_ the values to form a list which it returns. _mapconc_ compiles open.

Note that _mapcar_ creates a new list which is a mapping of the old list in that each element of the new list is the result of applying a function to the corresponding element on the original list. _mapconc_ is used when there are a *variable* number of elements (including none) to be inserted at each iteration, e.g. mapconc[X;(LAMBDA (Y) (AND Y (LIST Y)))] will make a list consisting of _x_ with all NILs removed, mapconc[X;(LAMBDA (Y) (AND (LISTP Y) Y))] will make a linear list consisting of all the lists on _x_, e.g. if applied to

((A B) C (D E F) (G) H I) will yield (A B D E F G).[2]

subset[mapx;mapfn1;mapfn2] applies mapfn1 to elements of mapx and returns a list of those elements for which this application is non-NIL, e.g., subset[(A B 3 C 4);NUMBERP] = (3 4). mapfn2 plays the same role as with map, mapc, et al. subset compiles open.

map2c[mapx;mapy;mapfn1;mapfn2] Identical to mapc except mapfn1 is a function of two arguments, and mapfn1[car[mapx];car[mapy]] is computed at each interation.[3] Terminates when *either* mapx or mapy are exhausted.

map2car[mapx;mapy;mapfn1;mapfn2] Identical to mapcar except mapfn1 is a function of two arguments and mapfn1[car[mapx];car[mapy]] is used to assemble the new list. Terminates when either mapx or mapy is exhausted.

Note: CLISP (Section 23) provides a more general and complete facility for expressing iterative statements, e.g. (FOR X IN Y COLLECT (CADR X) WHEN (NUMBERP (CAR X)) UNTIL (NULL X)).

---

[2] Note that since mapconc uses nconc to string the corresponding lists together, in this example, the original list will be clobbered, i.e. it would now be ((A B D E F G) C (D E F G) (G) H I). If this is an undesirable side effect, the functional argument to mapconc should return instead a top level copy, e.g. in this case, use (AND (LISTP Y) (APPEND Y)).

[3] mapfn2 is still a function of one argument, and is applied twice on each iteration; mapfn2[mapx] gives the new mapx, mapfn2[mapy] the new mapy. cdr is used if mapfn2 is not supplied, i.e., is NIL.

maprint[lst;file;left;right;sep;pfn;lispxprintflg]

is a general printing function. It cycles through lst applying pfn (or prin1 if pfn not given) to each element of lst. Between each application, maprint performs prin1 of sep, or " " if sep=NIL. If left is given, it is printed (using prin1) initially; if right is given it is printed (using prin1) at the end.

For example, maprint[x;NIL;%(;%)] is equivalent to prin1 for lists. To print a list with commas between each element and a final '.' one could use maprint[x;T;NIL;%.;%,].

If lispxprintflg = T, lispxprin1 is used for prin1 (see Section 22).

Mapdl,searchpdl See Section 12.

mapatoms See Section 5.

every, some, notevery, notany See Section 5.

Funarg

function is a function of two arguments, x, a function, and y a list of variables used freely by x. If y is not NIL, the *value* of function is an expression of the form (FUNARG x array), where array contains the bindings of the variables on y at the time the call to function was evaluated. funarg is not a function itself. Like LAMBDA and NLAMBDA, it has meaning and is specially recognized by INTERLISP only in the context of applying a function to arguments. In other words, the expression (FUNARG x array) is used exactly

like a function.[4] When a funarg is applied, the stack is modified so that the bindings contained in the array will be in force when x, the function, is called.[5]

For example, suppose a program wished to compute (FOO X (FUNCTION FIE)), and fie used y and z as free variables. If foo rebound y and z, fie would obtain the rebound values when it was applied from inside of foo. By evaluating instead (FOO X (FUNCTION FIE (Y Z))), foo would be called with (FUNARG FIE array) as its second argument, where array contained the bindings of y and z (at the time foo was called). Thus when fie was applied from inside of foo, it would 'see' the original values of y and z.

However, funarg is more than just a way of circumventing the clashing of variables. For example, a funarg expression can be returned as the value of a computation, and then used 'higher up', e.g., when the bindings of the variables contained in array were no longer on the stack. Furthermore, if the function in a funarg expression *sets* any of the variables contained in the array, the array itself (and only the array) will be changed. For example, suppose foo is defined as

(LAMBDA (LST FN) (PROG (Y Z) (SETQ Y &) (SETQ Z &) ... (MAPC LIST FN) ...))

and (FOO X (FUNCTION FIE (Y Z))) is evaluated. If one application of fie (by the mapc in foo) changes y and z, then the next application of fie will obtain the changed values of y and z resulting from the previous application of fie, since both applications of fie come from the exact same funarg object, and hence use the exact same array. The bindings of y and z bound inside of foo, and the bindings of y and z *above* foo would not be affected. In other words,

---

[4] LAMBDA, NLAMBDA, and FUNARG expressions are sometimes called 'function objects' to distinguish them from functions, i.e., literal atoms which have function definitions.

[5] The implementation of funarg is described in Section 12.

the variable bindings contained in array are a part of the function object, i.e., the funarg carries its environment with it.

Thus by creating a funarg expression with function, a program can create a function object which has updateable binding(s) associated with the object which last *between* calls to it, but are only accessible through that instance of the function. For example, using the funarg device, a program could maintain two different instances of the same random number generator in different states, and run them independently.

Example

If foo is defined as (LAMBDA (X) (COND ((ZEROP A) X) (T (MINUS X))) and fie as (LAMBDA NIL (PROG (A) (SETQ A 2) (RETURN (FUNCTION FOO)))). then if we perform (SETQ A 0), (SETQ FUM (FIE)), the value of fum is FOO, and the value of (APPLY* FUM 3) is 3, because the value of A at the time foo is called is 0.

However if fie were defined instead as
(LAMBDA NIL (PROG (A) (SETQ A 2) (RETURN (FUNCTION FOO (A))))), the value of fum would be (FUNARG FOO array) and so the value of (APPLY* FUM 3) would be -3, because the value of A seen by foo is the value A had when the funarg was created inside of fie, i.e. 2.

Index for Section 11

SECTION 12

VARIABLE BINDINGS AND PUSH DOWN LIST FUNCTIONS

A number of schemes have been used in different implementations of LISP for storing the values of variables. These include:

1. Storing values on an association list paired with the variable names.

2. Storing values on the property list of the atom which is the name of the variable.

3. Storing values in a special value cell associated with the atom name, putting old values on a pushdown list, and restoring these values when exiting from a function.

4. Storing values on a pushdown list.

The first three schemes all have the property that values are scattered throughout list structure space, and, in general, in a paging environment would require references to many pages to determine the value of a variable. This would be very undesirable in our system. In order to avoid this scattering, and possibly excessive drum references, we utilize a variation on the fourth standard scheme, usually only used for transmitting values of arguments to compiled functions; that is, we place these values on the pushdown list.[1] But

---

[1] Also called the stack.

since we use an interpreter as well as a compiler, the variable names must also be kept. The pushdown list thus contains pairs, each consisting of a variable name and its value. Each pair occupies one word or 'slot' on the pushdown list, with the name in the left half, i.e. cdr, and the value in the right half, i.e. car. The interpreter gets the value of a variable by searching back up the pushdown list looking for a 'slot' for which cdr is the name of the variable. car is then its value.

One advantage of this scheme is that the current top of the pushdown stack is usually in core, and thus drum references are rarely required to find the value of a variable. Free variables work automatically in a way similar to the association list scheme.

An additional advantage of this scheme is that it is completely compatible with compiled functions which pick up their arguments on the pushdown list from known positions, instead of doing a search. To keep complete compatibility, our compiled functions put the names of their arguments on the pushdown list, although they do not use them to reference variables. Thus, free variables can be used between compiled and interpreted functions with no *special* declarations necessary. The names on the pushdown list are also very useful in debugging, for they make possible a complete symbolic backtrace in case of error. Thus this technique, for a small extra overhead, minimizes drum references, provides symbolic debugging information, and allows completely free mixing of compiled and interpreted routines.

There are three pushdown lists used in INTERLISP: the first is called the parameter pushdown list, and contains pairs of variable names and values, and temporary storage of pointers; the second is called the control pushdown list, and contains function returns and other control information; and the third is called the number stack and is used for storing temporary partial results of numeric operations.

However, it is more convenient for the user to consider the push-down list as a single "list" containing the names of functions that have been entered but not yet exited, and the names and values of the corresponding variables. The multiplicity of pushdown lists in the actual implementation is for efficiency of operation only.

The Push-Down List and the Interpreter

In addition to the names and values of arguments for functions, information regarding partially-evaluated expressions is kept on the push-down list. For example, consider the following definition of the function fact (intentionally faulty):

```
(FACT
  [LAMBDA (N)
    (COND
      ((ZEROP N)
        L)
      (T (ITIMES N (FACT (SUB1 N])
```

In evaluating the form (FACT 1), as soon as fact is entered, the interpreter begins evaluating the implicit progn following the LAMBDA (see Section 4). The first function entered in this process is cond. cond begins to process its list of clauses. After calling zerop and getting a NIL value, cond proceeds to the next clause and evaluates T. Since T is true, the evaluation of the implicit progn that is the consequent of the T clause is begun (see Section 4). This requires calling the function itimes. However before itimes can be called, its arguments must be evaluated. The first argument is evaluated by searching the stack for the last binding of N; the second involves a recursive call to fact, and another implicit progn, etc.

Note that at each stage of this process, some portion of an expression has been evaluated, and another is awaiting evaluation. The output below illustrates

this by showing the state of the push-down list at the point in the computation of (FACT 1) when the unbound atom L is reached.

```
←FACT(1)
U.B.A.
(L BROKEN)
:BTV!

   *FORM* (BREAK1 L T L NIL #34047)
   #0 (L)

   #0 (((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))          1
COND

   *FORM* (COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))
   #0 ((COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N))))))    2

   N 0
FACT

   *FORM* (FACT (SUB1 N))
   #2 ITIMES
   #0 ((FACT (SUB1 N)))                                        3
   #0 1                                                        4
   *FORM* (ITIMES N (FACT (SUB1 N)))
   #0 ((ITIMES N (FACT (SUB1 N))))                             5

   #0 (((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))          6
COND

   *FORM* (COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))
   #0 ((COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N))))))    7

   N 1
FACT

**TOP**
```

Internal calls to <u>eval</u>, e.g., from <u>cond</u> and the interpreter, are marked on the push-down list by a special mark called an eval-blip. eval-blips are indicated by the appearance of (VAG 64) in the left-half, i.e. the variable name position, for that slot. They are printed by the backtrace as *FORM*. The genealogy of *FORM*'s is thus a history of the computation. Other temporary information is frequently recorded on the push-down list in slots for which the 'variable name' is (VAG 0), which prints as #0. In this example, this information consists of (1) the tail of a list of <u>cond</u> clauses, (2) the tail of an implicit <u>progn</u>, i.e., the definition of <u>fact</u>, (3) the tail of an argument

list, (4) the value of a previously evaluated argument, (5) the tail of a cond clause whose predicate evaluated to true, and (6) and (7) same as (1) and (2).

Note that a function is not actually entered and does not appear on the stack, until its arguments have been evaluated.[2] Also note that the #0 'bindings' comprise the actual working storage. In other words, in the above example, if a (lower) function changed the value of the binding at (1) (not recommended) the cond would continue interpreting the new binding as a list of cond clauses. Similarly, if (4) were changed, the new value would be given to itimes as its first argument after its second argument had been evaluated, and itimes was actually called.

The Pushdown List and Compiled Functions

Calls to compiled functions, and the bindings of their arguments, i.e. names and values, are handled in the same way as for interpreted functions (hence the compatibility between interpreted and compiled functions). However, compiled functions treat free variables in a special way that interpreted functions do not. Interpreted functions "look up" free variables when the variable is encountered, and may look up the same variable many times. However, compiled functions look up each free variable only once.[3] Whenever a compiled function is entered, the pushdown list is scanned and the most recent binding for each free variable used in the function is found (or if there is no binding, the value cell is obtained) and stored in the right half of a slot on the stack (an unboxed 0 is stored in the left half to distinguish this 'binding' from

---

[2] except for functions which do not have their arguments evaluated (although they themselves may call eval, e.g. cond).

[3] A list of all free variables is generated at compile time, and is in fact obtainable from the compiled definition. See Section 18.

ordinary bindings). Thus, following the bindings of their arguments, compiled functions store on the pushdown list pointers to the bindings for each free variable used in the function.

In addition to the pointers to free variable bindings, compiled functions differ from interpreted functions in the way they treat locally bound variables, i.e. progs and open lambdas. Whereas in interpreted functions progs and open lambdas are called in the ordinary way as functions, in compilation, progs and open lambdas disappear, although the variables bound by them are stored on the stack in the conventional manner so that functions called from inside them can reference the variables. These variables appear on the stack following the arguments to the compiled function (if any) and the free variable pointers (if any). The only way to determine dynamically what variables are bound locally by a compiled function is to search the stack from the first slot *beyond* the *last* argument to the function (which can be found with stknargs and stkarg described below), to the slot corresponding to the *first* argument of the *next* function. Any slots encountered that contain literal atoms in their left half are local bindings.

Pushdown List Functions

NOTE: Unless otherwise stated, for all pushdown list functions, pos is a position on the control stack. If pos is a literal atom other than NIL, (STKPOS pos 1) is used. In this case, if pos is not found, i.e., stkpos returns NIL, an ILLEGAL STACK ARG error is generated.

stkpos[fn;n;pos] Searches the control stack starting at pos for the nth occurrence of fn. Returns control stack

position of that fn if found,[4] else NIL. If n is positive, searches backward (normal usage). If n is negative, searches forward, i.e., down the control stack. For example, stkpos[FOO;-2;FIE] finds second call to FOO after (below) the last call to FIE. If n is NIL, 1 is used. If pos is NIL, the search starts at the current position. stkpos[] gives the current position.

stknth[n;pos] — Value is the stack position (control stack) of the nth function call relative to position pos. If pos is NIL, the top of stack is assumed for n > 0, and the current position is assumed for n < 0, i.e., stknth[-1] is the call before stknth, stknth[1] is the call to evalgt at the top level. Value of stknth is NIL if there is no such call - e.g., stknth[10000] or stknth[-10;stknth[5]].

fstknth[n;pos] — version of stknth that compiles open.

stkname[pos] — Value is the name of the function at control stack position pos. In this case, pos must be a real stack position, not an atom.

In summary, stkpos converts function names to stack positions, stknth converts numbers to stack positions, and stkname converts positions to function names.

---

[4] A stack position is a pointer to the corresponding slot on the control or parameter stack, i.e., the address of that cell. It prints as an unboxed number, e.g., #32002, and its type is 2 (Section 10).

Information about the variables bound at a particular function call can be obtained using the following functions:

stknargs[pos] — Value is the number of arguments bound by the function at position pos.

stkarg[n;pos] — Value is a pointer to the nth argument (named or not)[6] of the function at position pos, i.e., the value is a parameter stack position. car of this pointer gives the value of the binding, cdr the name. n=1 corresponds to the first argument at pos. n can be 0 or negative, i.e., stkarg[0;FOO] is a pointer to the slot immediately before the first argument to FOO, stkarg[-1;FOO] the one before that, etc.

fstkarg[n;pos] — version of stkarg that compiles open.

Note that the user can change (set) the value of a particular binding by performing an rplaca on the value of stkarg. Similarly, rplacd changes (sets) the name.

The value of stkarg is a position (slot) on the parameter stack. There is currently no analogue to stknth for the parameter stack. However, the parameter stack is a contiguous block of memory, so to obtain the slot previous to a given slot, perform vag[sub1[loc[slot]]]; to obtain the next slot perform vag[add1[loc[slot]]], i.e.

---

[6] Subrs do not store the names of their arguments.

stkarg[2;pos] = vag[add1[loc[stkarg[1;pos]]]].[6]

As an example of the use of stknargs and stkarg:

variables[pos] returns list of variables bound at pos.

can be defined by:

```
(VARIABLES
  [LAMBDA (POS)
    (PROG (N L)
          (SETQ N (STKNARGS POS))
      LP  (COND
            ((ZEROP N)
              (RETURN L)))
          (SETQ L (CONS (CDR (STKARG N POS))
                        L))
          (SETQ N (SUB1 N))
          (GO LP])
```

The counterpart of variables is also available.

stkargs[pos] Returns list of values of variables bound at pos.

The next three functions, stkscan, evalv, and stkeval all involve searching the parameter pushdown stack. For all three functions, pos may be a position on the control stack, i.e., a value of stkpos or stknth.[7] In this case, the search starts at stkarg[stknargs[pos];pos] i.e., it will include the arguments to the function at pos but not any locally bound variables. pos may also be a position on the *parameter* stack, in which case the search starts with, and includes that position. Finally, pos can be NIL, in which case the search starts with the current position on the parameter stack.

---

[6] See Section 13 for discussion of vag and loc.

[7] or a function name, which is equivalent to stkpos[pos;1] as described earlier.

stkscan[var;pos] — Searches backward on the parameter stack from pos for a binding of var. Value is the slot for that binding if found, i.e., a parameter stack position, otherwise var itself (so that in the case of literal atoms, car of stkscan is always the value of var).

evalv[var;pos] — car[stkscan[var;pos]],i.e., returns the value of the atom var as of position pos.

stkeval[pos;form] — is a more general evalv. It is equivalent to eval[form] at position pos, i.e., all *variables* evaluated in form, will be evaluated as of pos.[8]

Finally, we have two functions which clear the stacks:

retfrom[pos;value] — clears the stack back to the function at position pos, and effects a return from that function with value as its value.

reteval[pos;form] — clears the stack back to the function at position pos, *then* evaluates form and returns with its value to the next higher function. In other words, reteval[pos,form] is equivalent to retfrom[pos;stkeval[pos;form]].[9]

---

[8] However, any functions in form that specifically reference the stack, e.g., stkpos, stknth, retfrom, etc., 'see' the stack as it currently is. (See page 12.11-13 for description of how stkeval is implemented.)

[9] Provided form does not involve any stack functions, as explained in footnote 8.

We also have:

mapdl[mapdlfn;mapdlpos] starts at position mapdlpos (current if NIL), and applies mapdlfn to the function name at each pushdown position, i.e., to stkname[mapdlpos] until the top of stack is reached. Value is NIL. mapdlpos is updated at each iteration.

For example, mapdl[(LAMBDA (X) (AND (EXPRP X) (PRINT X)))] will print all exprs on the push-down list.
mapdl[(LAMBDA (X) (COND ((GREATERP (STKNARG MAPDLPOS) 2) (PRINT X] will print all functions of more than two arguments.

searchpdl[srchfn;srchpos] searches the pushdown list starting at position srchpos (current if NIL) until it finds a position for which srchfn applied to the name of the function called at that position is not NIL. Value is (NAME . position) if such a position is found, otherwise NIL. srchpos is updated at each iteration.

The Pushdown List and Funarg

The linear scan up the parameter stack for a variable binding can be interrupted by a special mark called a skip-blip appearing on the stack in a name position (See Figure 12-1). In the value position is a pointer to the position on the stack where the search is to be continued. This is what is used to make stkeval, page 12.10 work. It is also used by the funarg device (Section 11).

When a funarg is applied, INTERLISP puts a skip-blip on the parameter stack

with a pointer to the funarg array, and another skip-blip at the top of the funarg array pointing back to the stack. The effect is to make the stack look like it has a patch. The names and values stored in the funarg array will thus be seen before those higher on the stack. Similarly, setting a variable whose binding is contained in the funarg array will change only the array. Note however that as a consequence of this implementation, *the same instance of a funarg object cannot be used recursively.*

USE OF 'SKIPBLIPS'

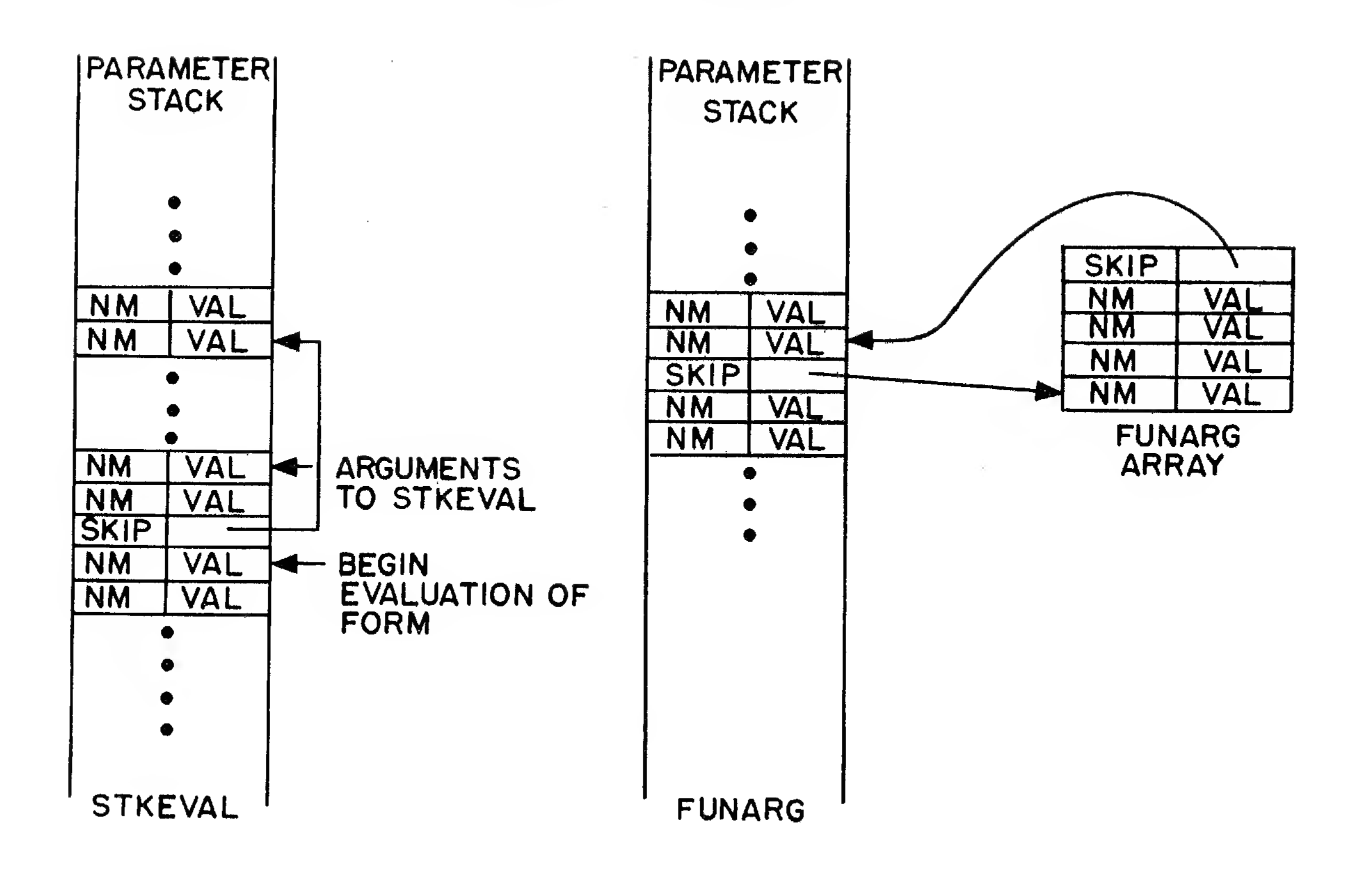


FIGURE 12-1

Index for Section 12

SECTION 13

NUMBERS AND ARITHMETIC FUNCTIONS

13.0 General Comments

There are three different types of numbers in INTERLISP: small integers, large integers, and floating point numbers.[1] Since a large integer or floating point number can be (in value) any 36 bit quantity (and vice versa), it is necessary to distinguish between those 36 bit quantities that represent large integers or floating point numbers, and other INTERLISP pointers. We do this by "boxing" the number, which is sort of like a special "cons": when a large integer or floating point number is created (via an arithmetic operation or by read), INTERLISP gets a new word from "number storage" and puts the large integer or floating point number into that word. INTERLISP then passes around the pointer to that word, i.e., the "boxed number", rather than the actual 36 bit quantity itself. Then when a numeric function needs the actual numeric quantity, it performs the extra level of addressing to obtain the "value" of the number. This latter process is called "unboxing". Note that unboxing does not use any storage, but that each boxing operation uses one new word of number storage. Thus, if a computation creates many large integers or floating point numbers, i.e., does lots of boxes, it may cause a garbage collection of large integer space, GC: 18, or of floating point number space, GC: 16.

---

[1] Floating point numbers are created by the read program when a . or an E appears in a number, e.g. 1000 is an integer, 1000. a floating point number, as are 1E3 and 1.E3. Note that 1000D, 1000F, and 1E3D are perfectly legal literal atoms.

13.1 Integer Arithmetic

Small Integers

Small integers are those integers for which smallp is true, currently integers whose absolute value is less than 1536. Small integers are boxed by offsetting them by a constant so that they overlay an area of INTERLISP's address space that does not correspond to any INTERLISP data type. Thus boxing small numbers does not use any storage, and furthermore, each small number has a unique representation, so that eq may be used to check equality. Note that eq should not be used for large integers or floating point numbers, e.g., eq[2000;add1[1999]] is NIL! eqp or equal must be used instead.

Integer Functions

All of the functions described below work on integers. Unless specified otherwise, if given a floating point number, they first convert the number to an integer by truncating the fractional bits, e.g., iplus[2.3;3.8]=5; if given a non-numeric argument, they generate an error, NON-NUMERIC ARG.

It is important to use the *integer* arithmetic functions, whenever possible, in place of the more general arithmetic functions which allow mixed floating point and integer arithmetic, e.g., iplus vs plus, igreaterp vs greaterp, because the integer functions compile open, and therefore run faster than the general arithmetic functions, and because the compiler is "smart" about eliminating unnecessary boxing and unboxing. Thus, the expression (IPLUS (IQUOTIENT (ITIMES N 100) M) (ITIMES X Y)) will compile to perform only one box, the outer one, and the expression (IGREATERP (IPLUS X Y) (IDIFFERENCE A B)) will compile to do no boxing at all.

Note that the PDP-10 is a 36 bit machine, so that all integers are between -2↑35 and 2↑35-1.[2] Adding two integers which produce a result outside this range causes overflow, e.g., 2↑34 + 2↑34.

The procedure on overflow is to return the largest possible integer, i.e. 2↑35 - 1.[3]

iplus[x_1;x_2;...;x_n]	$x_1 + x_2 + \ldots + x_n$
iminus[x]	- x
idifference[x;y]	x - y
add1[x]	x + 1
sub1[x]	x - 1
itimes[x_1;x_2;...;x_n]	the product of $\underline{x}_1$,$\underline{x}_2$,...$\underline{x}$n
iquotient[x;y]	x/y truncated, e.g., iquotient[3;2]=1, iquotient[-3,2]=-1
iremainder[x;y]	the remainder when $\underline{x}$ is divided by $\underline{y}$, e.g., iremainder [3;2]=1
igreaterp[x;y]	T if x > y; NIL otherwise

---

[2] Approximately 34 billion

[3] If the overflow occurs by trying to create a negative number of too large a magnitude, -2↑35 is used instead of 2↑35-1.

ilessp[x;y] T is x < y; NIL otherwise

zerop[x] defined as eq[x;0].

Note that zerop should not be used for floating point numbers because it uses eq. Use eqp[x;0] instead.

minusp[x] T if x is negative; NIL otherwise. Does not convert x to an integer, but simply checks sign bit.

eqp[n;m] T if n and m are eq, or equal numbers, NIL otherwise. (eq may be used if n and m are known to be small integers.) eqp does not convert n and m to integers, e.g., eqp[2000;2000.3]=NIL, but it can be used to compare an integer and a floating point number, e.g., eqp[2000;2000.0]=T. eqp does *not* generate an error if n or m are not numbers.

smallp[n] T if n is a small integer, else NIL. smallp does *not* generate an error if n is not a number.

fixp[x] x if x is an integer, else NIL. Does *not* generate an error if x is not a number.

fix[x] Converts x to an integer by truncating fractional bits, e.g., fix[2.3] = 2, fix[-1.7] = -1. If x is already an integer, fix[x]=x and doesn't use any storage.[4]

---

[4] Since FIX is also a lispx command (Section 22), typing FIX directly to lispx will not cause the function fix to be called.

$logand[x_1;x_2;...;x_n]$ — lambda no-spread, value is logical and of all its arguments, as an integer, e.g., logand[7;5;6]=4.

$logor[x_1;x_2;...;x_n]$ — lambda no-spread, value is the logical or of all its arguments, as an integer, e.g., logor[1;3;9]=11.

$logxor[x_1;x_2;...;x_n]$ — lambda no-spread, value is the logical exclusive or of its arguments, as an integer, e.g., logxor[11;5] = 14, logxor[11;5;9] = logxor[14;9] = 7.

lsh[n;m] — (arithmetic) left shift, value is n*2↑m,i.e., n is shifted left m places. n can be positive or negative. If m is negative, n is shifted *right* -m places.

rsh[n;m] — (arithmetic) right shift, value is n*2↑-m, i.e., n is shifted right m places. n can be positive or negative. If m is negative, n is *left* -m places.

llsh[n;m] — logical left shift. On PDP-10, llsh is equivalent to lsh.

lrsh[n;m] — logical right shift.

The difference between a logical and arithmetic right shift lies in the treatment of the sign bit for negative numbers. For arithmetic right shifting of negative numbers, the sign bit is propagated, i.e., the value is a negative number. For logical right shift, zeroes are propagated. Note that shifting (arithmetic) a negative number 'all the way' to the right yields -1, not 0.

13.2 Floating Point Arithmetic

All of the functions described below work on floating point numbers. Unless specified otherwise, if given an integer, they first convert the number to a floating point number, e.g., fplus[1;2.3] = fplus[1.0;2.3] = 3.3; if given a non-numeric argument, they generate an error, NON-NUMERIC ARG.

The largest floating point number is 1.7014118E38, the smallest positive (non-zero) floating point number is 1.4693679E-39. The procedure on overflow is the same as for integer arithmetic. For underflow, i.e. trying to create a number of too small a magnitude, the value will be 0.

fplus[x_1;x_2;...x_n] — $x_1 + x_2 + \ldots + x_n$

fminus[x] — - x

ftimes[x_1;x_2;...;x_n] — $x_1 * x_2 * \ldots * x_n$

fquotient[x;y] — x/y

fremainder[x;y] — the remainder when *x* is divided by *y*, e.g., fremainder[1.0;3.0]= 3.72529E-9.

minusp[x] — T if *x* is negative; NIL otherwise. Works for both integers and floating point numbers.

eqp[x;y] — T if *x* and *y* are *eq*, or equal numbers. See discussion page 13.4.

fgtp[x;y] — T if x > y, NIL otherwise.

floatp[x] — is x if x is a floating point number; NIL otherwise. Does *not* give an error if x is not a number.

Note that if numberp[x] is true, then either fixp[x] or floatp[x] is true.

float[x] — Converts x to a floating point number, e.g., float[0] = 0.0.

13.3 Mixed Arithmetic

The functions in this section are 'contagious floating point arithmetic' functions, i.e., if any of the arguments are floating point numbers, they act exactly like floating point functions, and float all arguments, and return a floating point number as their value. Otherwise, they act like the integer functions. If given a non-numeric argument, they generate an error, NON-NUMERIC ARG.

plus[x_1;x_2;...;x_n] — $x_1 + x_2 + ... + x_n$

minus[x] — $- x$

difference[x;y] — $x - y$

times[x_1;x_2;...;x_n] — $x_1 * x_2 * ... * x_n$

quotient[x;y] — if x and y are both integers, value is iquotient[x;y], otherwise fquotient[x;y].

remainder[x;y] — if x and y are both integers, value is iremainder[x;y], otherwise fremainder[x;y].

greaterp[x;y] — T if x > y, NIL otherwise.

lessp[x;y] — T if x < y, NIL otherwise.

abs[x] — *x* if x > 0, otherwise -x. *abs* uses *greaterp* and *minus*, (not *igreaterp* and *iminus*).

13.4 Special Functions

These functions are all "borrowed" from the FORTRAN library and handcoded in INTERLISP via ASSEMBLE by J. W. Goodwin. They utilize a power series expansion and their values are (supposed to be) 27 bits accurate, e.g., sin[30]=.5 exactly.

expt[m;n] — value is m↑n. If *m* is an integer and *n* is a positive integer, value is an integer, e.g, expt[3;4]=81, otherwise the value is a floating point number. If *m* is negative and *n* fractional, an error is generated.

sqrt[n] — value is a square root of *n* as a floating point number. *n* may be fixed or floating point. Generates an error if *n* is negative. sqrt[n] is about twice as fast as expt[n;.5]

log[x] — value is natural logarithm of *x* as a floating point number. *x* can be integer or floating point.

antilog[x] — value is floating point number whose logarithm is *x*. *x* can be integer or floating point, e.g., antilog[1] = e = 2.71828...

sin[x;radiansflg]	x in degrees unless radiansflg=T. Value is sine of x as a floating point number.
cos[x;radiansflg]	Similar to sin.
tan[x;radiansflg]	Similar to sin.
arcsin[x;radiansflg]	x is a number between -1 and 1 (or an error is generated). The value of arcsin is a floating point number, and is in degrees unless radiansflg=T. In other words, if arcsin[x;radiansflg]=z then sin[z;radiansflg]=x. The range of the value of arcsin is -90 to +90 for degrees, $-\pi/2$ to $\pi/2$ for radians.
arccos[x;radiansflg]	Similar to arcsin. Range is 0 to 180, 0 to π.
arctan[x;radiansflg]	Similar to arcsin. Range is 0 to 180, 0 to π.
rand[lower;upper]	Value is a pseudo-random number between lower and upper inclusive, i.e. rand can be used to generate a sequence of random numbers. If both limits are integers, the value of rand is an integer, otherwise it is a floating point number. The algorithm is completely deterministic, i.e. given the same initial state, rand produces the same sequence of values. The internal state of rand is initialized using the function randset described below, and is stored on the free variable randstate.

randset[x] — Value is internal state of rand after randset has finished operating, as a dotted pair of two integers. If x=NIL, value is current state. If x=T, randstate is initialized using the clocks. Otherwise, x is interpreted as a previous internal state, i.e. a value of randset, and is used to reset randstate. For example,

1. (SETQ OLDSTATE (RANDSET))
2. Use rand to generate some random numbers.
3. (RANDSET OLDSTATE)
4. rand will generate same sequence as in 2.

13.5 Reusing Boxed Numbers - SETN

rplaca and rplacd provide a way of cannibalizing list structure for reuse in order to avoid making new structure and causing garbage collections.[5] This section describes an analogous function for large integers and floating point numbers, setn. setn is used like setq, i.e., its first argument is considered as quoted, its second is evaluated. If the current value of the variable being set is a large integer or floating point number, the new value is deposited into that word in number storage, i.e., no new storage is used.[6] If the current value is *not* a large integer or floating point number, e.g., it can be NIL, setn operates exactly like setq, i.e., the large integer or floating point number is boxed, and the variable is set. This eliminates initialization of the variable.

---

[5] This technique is frowned upon except in well-defined, localized situations where efficiency is paramount.

[6] The second argument to setn must always be a number or a NON-NUMERIC ARG error is generated.

setn will work interpretively, i.e., reuse a word in number storage, but will not yield any savings of storage because the boxing of the second argument will still take place, when it is evaluated. The elimination of a box is achieved only when the call to setn is compiled, since setn compiles open, and does not perform the box if the old value of the variable can be reused.

Caveats concerning use of SETN

There are three situations to watch out for when using setn. The first occurs when the same variable is being used for floating point numbers and large integers. If the current value of the variable is a floating point number, and it is reset to a large integer, via setn, the large integer is simply deposited into a word in floating point number storage, and hence will be interpreted as a floating point number. Thus,

```
←(SETQ FOO 2.3)
2.3
←(SETN FOO 10000)
2.189529E-43
```

Similarly, if the current value is a large integer, and the new value is a floating point number, equally strange results occur.

The second situation occurs when a setn variable is reset from a large integer to a small integer. In this case, the small integer is simply deposited into large integer storage. It will then print correctly, and function arithmetically correctly, but it is *not* a small integer, and hence will not be eq to another integer of the same value, e.g.,

```
←(SETQ FOO 10000)
10000
←(SETN FOO 1)
1
←(IPLUS FOO 5)
6
←(EQ FOO 1)
NIL
←(SMALLP FOO)
NIL
```

In particular, note that zerop will return NIL even if the variable is equal to 0. Thus a program which begins with FOO set to a large integer and counts it down by (SETN FOO (SUB1 FOO)) must terminate with (EQP FOO 0), not (ZEROP FOO).

Finally, the third situation to watch out for occurs when you want to save the current value of a setn variable for later use. For example, if FOO is being used by setn, and the user wants to save its current value on FIE, (SETQ FOO FIE) is not sufficent, since the next setn on FOO will also change FIE, because its changes the word in number storage pointed to by FOO, and hence pointed to by FIE. The number must be copied, e.g., (SETQ FIE (IPLUS FOO)), which sets FIE to a new word in number storage.

setn[var;x] — nlambda function like setq. var is quoted, x is evaluated, and its value must be a number. var will be set to this number. If the current value of var is a large integer or floating point number, that word in number storage is cannibalized. The value of setn is the (new) value of var.

13.6 Box and Unbox

Some applications may require that a user program explicitly perform the boxing and unboxing operations that are usually implicit (and invisible) to most

programs. The functions that perform these operations are loc and vag respectively. For example, if a user program executes a TENEX JSYS using the ASSEMBLE directive, the value of the ASSEMBLE expression will have to be boxed to be used arithmetically, e.g., (IPLUS X (LOC (ASSEMBLE --))). It must be emphasized that

Arbitrary unboxed numbers should not be passed around as ordinary values because they can cause trouble for the garbage collector.

For example, suppose the value of x were 150000, and you created (VAG X), and this just *happened* to be an address on the free storage list] The next garbage collection could be disastrous. For this reason, the function vag must be used with extreme caution when its argument's range is not known.

One place where vag is safe to use is for performing computations on stack positions, which are simply *addresses* of the corresponding positions (cells) on the stack. To treat these addresses as *numbers*, the program must first box them. Conversely, to convert numbers to corresponding stack positions, the program must unbox them. Thus, suppose x were the value of stkarg, i.e., x corresponds to a position on the parameter stack. To obtain the next position on the stack, the program must compute (VAG (ADD1 (LOC X))). Thus if x were #32002,[7] (LOC X) would be 32002Q,[8] (ADD1 (LOC X)) would be 32003Q, and (VAG (ADD1 (LOC X))) would be #32003.

Note that rather than starting with a number, and unboxing it to obtain its numeric quantity, here we started with an address, i.e., a 36 bit quantity, and

---

[7] An INTERLISP pointer (address) which does not correspond to the address of a list structure, or an atom, or a number, or a string, is printed as #n, n given in octal.

[8] Q following a number means the numeric quantity is expressed in octal.

wishing to treat it as a number, boxed it. For example, loc of an atom, e.g., (LOC (QUOTE FOO)), treats the atom as a 36 bit quantity, and makes a number out of it. If the address of the atom FOO were 125000, (LOC (QUOTE FOO)) would be 125000, i.e. the location of FOO. It is for this reason that the box operation is called loc, which is short for location.[9]

Note that FOO does not print as #364110 (125000 in octal) because the print routine recognizes that it is an atom, and therefore prints it in a special way, i.e. by printing the individual characters that comprise it. Thus (VAG 125000) would print as FOO, and *would be* in fact FOO.

loc[x] — Makes a number out of x, i.e., returns the location of x.

vag[x] — The inverse of loc. x must be a number; the value of vag is the unbox of x.

The compiler eliminates extra vag's and loc's for example (IPLUS X (LOC (ASSEMBLE --))) will not box the value of the ASSEMBLE, and then unbox it for the addition.

---

[9] vag is an abbreviation of value get.

Index for Section 13

Page Numbers

SECTION 14

INPUT/OUTPUT FUNCTIONS

14.1 Files

All input/output functions in INTERLISP can specify their source/destination file with an optional extra argument which is the name of the file. This file must be opened as specified below. If the extra argument is not given (has value NIL), the file specified as "primary" for input (output) is used. Normally these are both T, for teletype input and output. However, the primary input/output file may be changed by

input[file][1] — Sets file as the primary input file. Its value is the name of the old primary input file.

input[] returns current primary input file, which is not changed.

output[file] — Same as input except operates on primary output file.

Any file which is made primary must have been previously opened for input/output, except for the file T, which is always open.

---

[1] The argument name file is used for tutorial purposes only. The arguments to all subrs are U, V, and W as described in arglist, Section 8.

infile[file] — Opens <u>file</u> for input, and sets it as the primary input file.[2] The value of <u>infile</u> is the previous primary input file. If <u>file</u> is already open, same as input[file]. Generates a FILE WON'T OPEN error if <u>file</u> won't open, e.g., <u>file</u> is already open for *output*.

outfile[file] — Opens <u>file</u> for output, and sets it as the primary output file.[3] The value of <u>outfile</u> is the previous primary output file. If <u>file</u> is already open, same as output[file]. Generates a FILE WON'T OPEN error if <u>file</u> won't open, e.g., if <u>file</u> is already open for *input*.

For all input/output functions, <u>file</u> follows the TENEX conventions for file names, i.e. <u>file</u> can be prefixed by a directory name enclosed in angle brackets, can contain alt-modes or control-F's, and can include suffixes and/or version numbers. Consistent with TENEX, when a file is opened for input and no version number is given, the highest version number is used. Similarly, when a file is opened for output and no version number is given, a new file is created with a version number one higher than the highest one currently in use with that file name.

Regardless of the file name given to the INTERLISP function that opened the

---

[2] To open <u>file</u> without changing the primary input file, perform input[infile[file]].

[3] To open <u>file</u> without changing the primary output file, perform output[outfile[file]].

file, INTERLISP maintains only full TENEX file names[4] in its internal table of open files and any function whose value is a file name always returns a full file name, e.g. openp[FOO]=FOO.;3. Whenever a file argument is given to an i/o function, INTERLISP first checks to see if the file is in its internal table. If not, INTERLISP executes the appropriate TENEX JSYS to "recognize" the file. If TENEX does not successfully recognize the file, a FILE NOT FOUND error is generated.[5] If TENEX does recognize the file, it returns to INTERLISP the full file name. Then, INTERLISP can continue with the indicated operation. If the file is being opened, INTERLISP opens the file and stores its (full) name in the file table. If it is being closed, or written to or read from, INTERLISP checks its internal table to make sure the file is open, and then executes the corresponding operation.

Note that each time a full file name is *not* used, INTERLISP must call TENEX to recognize the name. Thus if repeated operations are to be performed, it is considerably more efficient to obtain the full file name once, e.g. via infilep or outfilep. Also, note that recognition by TENEX is performed on the user's entire directory. Thus, even if only one file is open, say FOO.;1, F$ (F altmode) will not be recognized if the user's directory also contains the file FIE.;1. Similarly, it is possible for a file name that was previously recognized to become ambiguous. For example, a program performs infile[FOO], opening FOO.;1, and reads several expressions from FOO. Then the user types control-C, creates a FOO.;2 and reenters his program. Now a call to read giving it FOO as its file argument will generate a FILE NOT OPEN error, because TENEX will recognize FOO as FOO.;2.

---

[4] i.e. name, extension, and version, plus directory name if it differs from connected directory.

[5] except for infilep, outfilep and openp, which in this case return NIL.

infilep[file] — Returns full file name of file if recognized by TENEX, NIL otherwise. The full file name will contain a directory field only if the directory differs from the currently attached directory. Recognition is in input context, i.e. if no version number is given, the highest version number is returned.

infilep and outfilep do not open any files, or change the primary files; they are pure predicates.

outfilep[file] — Similar to infilep, except recognition is in output context, i.e. if no version number is given, a version number one higher than the highest version number is returned.

closef[file] — Closes file. Generates an error, FILE NOT OPEN, if file not open. If file is NIL, it attempts to close the primary input file if other than teletype. Failing that, it attempts to close the primary output file if other than teletype. Failing both, it returns NIL. If it closes any file, it returns the name of that file. If it closes either of the primary files, it resets that primary file to teletype.

closeall[] — Closes all open files (except T). Value is a list of the files closed.

openp[file;type] — If type=NIL, value is file (full name) if file is open either for reading or for writing. Otherwise value is NIL.

If type is INPUT or OUTPUT, value is file if open for corresponding type, otherwise NIL. If type is BOTH, value is file if open for both input *and* output, (See iofile, page 14.6) otherwise NIL.

Note: the value of openp is NIL if file is not recognized, i.e. openp does not generate an error.

openp[] returns a list of all files open for input or output, excluding T.

Addressable Files

For most applications, files are read starting at their beginning and proceeding sequentially, i.e. the next character read is the one immediately following the last character read. Similarly, files are written sequentially. A program need not be aware of the fact that there is a file pointer associated with each file that points to the location where the next character is to be read from or written to, and that this file pointer is automatically advanced after each input or output operation. This section describes a function which can be used to *reposition* the file pointer, thereby allowing a program to treat a file as a large block of auxiliary storage which can be access randomly.[6] For example, one application might involve writing an expression at the *beginning* of the file, and then reading an expression from a specified point in its

---

[6] Random access means that any location is as quickly accessible as any other. For example, an array is randomly accessible, but a list is not, since in order to get to the nth element you have to sequence through the first n-1 elements.

middle.[7]

A file used in this fashion is much like an array in that it has a certain number of addressable locations that characters can be put into or taken from. However, unlike arrays, files can be enlarged. For example, if the file pointer is positioned at the end of a file and anything is written, the file "grows." It is also possible to position the file pointer *beyond* the end of file and then to write.[8] In this case, the file is enlarged, and a "hole" is created, which can later be written into. Note that this enlargement only takes place at the *end* of a file; it is not possible to make more room in the middle of a file. In other words, if expression A begins at positon 1000, and expression B at 1100, and the program attempts to overwrite A with expression C, which is 200 characters long, part of B will be clobbered.

iofile[file] — Opens file for both input and output. Value is file. Does not change either primary input or primary output. If no version number is given, default is same as for infile, i.e. highest version number.

sfptr[file;address] — Sets file pointer for file to address.[9] Value is

---

[7] This particular example requires the file be open for *both* input and output. This can be achieved via the function iofile described below. However, random file input or output can be performed on files that have been opened in the usual way by infile or outfile.

[8] If the program attempts to *read* beyond the end of file, an END OF FILE error occurs.

[9] TENEX uses byte addressing; the address of a character (byte) is the number of characters (bytes) that precede it in the file, i.e., 0 is the address of the beginning of the file. However, the user should be careful about computing the space needed for an expression, since end-of-line is represented as two characters in a file, but nchars only counts it as one.

old setting. address=-1 corresponds to the end of file.[10]

If address=NIL, sfptr returns the current value of file pointer without changing it.

filepos[x;file;start;end;skip;tail][11] Searches file for x a la strpos (Section 10). Search begins at start (or if start=NIL, the current position of file pointer), and goes to end (or if end=NIL, to the end of file). Value is *address* of start of match, or NIL if not found. skip can be used to specify a character which matches any character in the file. If tail is T, and the search is successful, the value is the address of the first character *after* the sequence of characters corresponding to x, instead of the starting address of the sequence. In either case, the file is left so that the next i/o operation begins at the address returned as the value of filepos.

---

[10] Note: if a file is opened for output only, either by outfile, or openf[file;100000q] (see page 14.8), TENEX assumes that one intends to write a new or different file, even if a version number was specified and the corresponding file already exists. Thus, sfptr[file;-1] will set the file pointer to 0. If a file is opened for both reading and writing, either by iofile or openf[file;300000q], TENEX assumes that there might be material on the file that the user intends to read. Thus, the initial file pointer is the beginning of the file, but sfptr[file;-1] will set it to the end of the file. Note that one can also open a file for appending by openf[file;20000q]. In this case, the file pointer right after opening is set to the end of the existing file. Thus, a write will automatically add material at the end of the file, and an sfptr is unnecessary.

[11] filepos was written by J.W. Goodwin.

Openf

openf[file;x] — opens file. x is a number whose bits specify the access and mode for file, i.e. x corresponds to the second argument to the TENEX JSYS OPENF (see JSYS Manual). Value is full name of file.

openf permits opening a file for read, write, execute, or append, etc. and allows specification of byte size, i.e. a byte size of 36 enables reading and writing of full words. openf does not affect the primary input or output file settings, and does not check whether the file is already open - i.e. the same file can be opened more than once, possibly for different purposes.[12] openp will work for files opened with openf.

The first argument to openf can also be a number, which is then interpreted as JFN. This results in a more efficient call to openf, and can be signficant if the user is making frequent calls to openf, e.g. switching byte sizes.

JFN Functions[13]

JFN stands for job file number. It is an integral part of the TENEX file system and is described in [Murl], and in somewhat more detail in the TENEX JSYS manual. The following function can be used to obtain the JFN for an already opened file.

opnjfn[file] — returns the JFN for file. If file not open, generates a FILE NOT OPEN error.

---

[12] The "thawed" bit in x permits opening a file that is already open.

[13] The JFN functions were written by J.W. Goodwin.

Example: to write a byte on a file

```
[DEFINEQ (BOUT
     (LAMBDA (FILE BYTE)
     (LOC (ASSEMBLE NIL
              (CQ (VAG BYTE))
              (PUSH NP , 1)
              (CQ (VAG (OPNJFN FILE)))
              (POP NP , 2)
              (JSYS 51Q)
              (MOVE 1 , 2)]
```

or to read a byte from a file

```
[DEFINEQ (BIN
     (LAMBDA (FILE)
        (LOC (ASSEMBLE NIL
              (CQ (VAG (OPNJFN FILE)))
              (JSYS 50Q)
              (MOVE 1 , 2]
```

Making BIN and BOUT substitution macros can save boxing and unboxing in compiled code.

The following functions are available for direct manipulation of JFN's:

gtjfn[file;ext;v;flags] — sets up a 'long' call to GTJFN (see JSYS manual). _file_ is a file name possibly containing control-F and/or alt-mode. _ext_ is the default extension, _v_ the default version (overriden if _file_ specifies extension/version, e.g. FOO.COM;2). _flags_ is as described on page 17, section 2 of JSYS manual. _file_ and _ext_ may be strings or atoms; _v_ and _flags_ must be numbers. Value is JFN, or NIL on errors.

rljfn[jfn] — releases _jfn_. rljfn[-1] releases all JFN's which do not specify open files. Value of _rljfn_ is T.

jfns[jfn;ac3] — converts _jfn_ (a small number) to a file name. _ac3_

is either NIL, meaning format the file name as would openp or other INTERLISP file functions, or else is a number, meaning format according to JSYS manual. The value of jfns is atomic except where enough options are specified by ac3 to exceed atom size (≥ 100 characters). In this case, the value is returned as a string.

14.2 Input Functions

Most of the functions described below have an (optional) argument file which specifies the name of the file on which the operation is to take place. If that argument is NIL, the primary input file will be used.

Note: in all INTERLISP symbolic files, end-of-line is indicated by the characters carriage-return and line-feed in that order. Accordingly, on input from files, INTERLISP will skip all line-feeds which immediately follow carriage-returns.[14] *On input from teletype, INTERLISP will echo a line-feed whenever a carriage-return is input.*

For all input functions except readc and peekc, when reading from the teletype, control-A erases the last character typed in, echoing a \ and the erased character. Control-A will not backup beyong the last carriage return. Typing control-Q causes INTERLISP to print ## and clear the input buffer, i.e. erase the entire line back to the last carriage-return. When reading from a file, and an end of file is encountered, all input functions close the file and generate an error, END OF FILE.

read[file;flg]

Reads one S-expression from file. Atoms are delimited by parentheses, brackets, double quotes, spaces, and carriage-returns. To input an atom which contains one of these syntactic delimiters, preceded the delimiter by the escape character %, e.g. AB%(C, is the atom AB(C, %% is the atom %.

---

[14] Actually, INTERLISP skips the next character after a carriage-return without looking at it at all.

Strings are delimited by double quotes. To input a string containing a double quote or a %, precede it by %, e.g. "AB%"C" is the string AB"C. Note that % can always be typed even if next character is not 'special', e.g. %A%B%C is read as ABC.

If an atom is interpretable as a number, read will create a number, e.g. 1E3 reads as a floating point number, 1D3 as a literal atom, 1.0 as a number, 1,0 as a literal atom, etc. Note that an integer can be input in octal by terminating it with a Q, e.g. 17Q and 15 read in as the same integer. The setting of radix, page 14.22, determines how integers are printed, i.e. with or without Q's.

When reading from the teletype, all input is line-buffered to enable the action of control-Q.[15] *Thus no characters are actually seen by the program until a carriage-return is typed. However, for reading by read or uread, when a matching right parenthesis is encountered, the effect is the same as though a carriage return were typed, i.e. the characters are transmitted. To indicate this, INTERLISP also prints a carriage-return line-feed on the teletype.*

flg=T suppresses the carriage-return normally typed by read following a matching right parenthesis. (However, the characters are still given to read - i.e. the user does not have to type the carriage return himself.)

ratom[file] Reads in one atom from file. Separation of atoms

---

[15] Unless control[T] has been performed (page 14.24).

is defined by action of setsepr and setbrk described below. % is also an escape character for ratom, and the remarks concerning control-A, control-Q, and line-buffering also apply.

If the characters comprising the atom would normally be interpreted as a number by read, that number is also returned by ratom. Note however that ratom takes no special action for " whether or not it is a break character, i.e. ratom never makes a string.

The purpose of ratom, rstring, setbrk, and setsepr is to allow the user to write his own read program without having to resort to reading character by character and then calling pack to make atoms. The function uread (page 14.15) is available if the user wants to handle input as read does, i.e. same action on parentheses, double quotes, square brackets, dot, spaces, and carriage-return, but in addition, to split atoms that contain certain characters, as specified by setbrk and setsepr.

rstring[file] — Reads in one string from file, terminated by next break or separator character. Control-A, control-Q, and % have the same effect as with ratom.

Note that the break or separator character that terminates a call to ratom or rstring is not read by that call, but remains in the buffer to become the first character seen by the next reading function that is called.

ratoms[a;file] — Calls ratom repeatedly until the atom a is read. Returns a list of atoms read, not including a.

setsepr[lst;flg] — Set separator characters. Value is NIL.

setbrk[lst;flg] — Set break characters. Value is NIL.

For both setsepr and setbrk,lst is a list of character codes, flg determines the action of setsepr/setbrk as follows:

NIL clear out old tables and reset.

0 clear out only those characters in lst - i.e. this provides an unsetsepr and unsetbrk.

1 add characters in lst to corresponding table.

Characters specified by setbrk will delimit atoms, and be returned as separate atoms themselves by ratom.[16] Characters specified by setsepr will be ignored and serve only to separate atoms. For example, if $ was a break character and] a separator character, the input stream ABC]]DEF$GH]$$ would be read by 6 calls to ratom returning respectively ABC, DEF, $, GH, $, $.

Note that the action of % is not affected by setsepr or setbrk. To defeat the action of % use escape[], as described below.

The elements of lst may also be characters e.g. setbrk[(%(%))] has the same effect as setbrk[(40 41)]. Note however that the 'characters' 1,2...9,0 will be interpreted as character *codes* because they are numbers.

Initially, the break characters are [ ] ( ) and " and the separator characters are space, tab, carriage-return, line-feed, end-of-line, and form-feed. (Note that . is not a break or separator character.) setbrk[T] sets the break characters to their initial settings, and setsepr[T] does the same for the separator characters.

getsepr[] Value is a list of separator character codes.

---

[16] but have no effect whatsoever on the action of read.

getbrk[ ] — Value is a list of break character codes.

escape[flg] — If flg=NIL, makes % act like every other character. Normal setting is escape[T].

The value of escape is the previous setting.

ratest[x] — If x = T, ratest returns T if a separator was encountered immediately prior to the last atom read by ratom, NIL otherwise.

If x = NIL, ratest returns T if last atom read by ratom or read was a break character, NIL otherwise.

If x = 1, ratest returns T if last atom read (by read or ratom) contained a % (as an escape character, e.g., %[or %A%B%C), NIL otherwise.

readc[file] — Reads the next character, including %, ", etc. Value is the character. Action of readc is subject to line-buffering, i.e. readc will not return a value until the line has been terminated even if a character has been typed. Thus, control-A and control-Q will have their usual effect. If control[T] has been executed (page 14.24), defeating line-buffering, readc will return a value as soon as a character is typed. In addition, if control-A or control-Q are typed, readc will return them as values.

peekc[file;flg]

Value is the next character, but does not actually read it, i.e. remove it from the buffer. If flg=NIL, peekc is not subject to line-buffering, i.e. it returns a value as soon as a character has been typed. If flg=T, peekc waits until the line has been terminated before returning its value. This means that control-A and control-Q will be able to perform their usual editing functions.

lastc[file]

Value is last character read from file.

uread[file;flg]

(for user read). Same as read except it uses separator and break characters set by setsepr and setbrk. This function is useful for reading in list structure in the normal way, while splitting atoms containing special characters. Thus with space a separator character, and break characters of ( ) . and ' the input stream (IT'S EASY.) is read by uread as the list (IT ' S EASY %.)

Note that ( ) [ ] and " must be included in the break characters if uread is to take special action on them, i.e. assemble lists and make strings.

flg=T suppresses carriage-return normally typed following a matching right parentheses. See page 14.11.

Note: read, ratom, ratoms, peekc, readc, and uread all wait for input if there is none. The only way to test whether or not there is input is to use readp.

readp[file]

Value is T if there is anything in the input buffer of file, NIL otherwise (not particularly meaningful for file other than T). Note that because of line-buffering, readp may return T, indicating there is input in the buffer, but read may still have to wait.

readline[][17]

reads a line from the teletype, returning it as a list. If readp[T] is NIL, readline returns NIL. Otherwise it reads expressions, using read,[18] until it encounters either:

(1) a carriage-return (typed by the user) that is not preceded by any spaces, e.g.

A B C↵

and readline returns (A B C)

(2) a list terminating in a ']', in which case the list is included in the value of readline, e.g. A B (C D] and readline returns (A B (C D)).

---

[17] Readline actually has two arguments for use by the system, but the user should consider it as a function of no arguments.

[18] Actually, readline performs (APPLY* LISPXREADFN T), as described in Section 22. lispxreadfn is initially READ.

(3) an unmatched right parentheses or right square bracket, which is not included in the value of readline, e.g.

```
A B C]
```

and readline returns (A B C).

In the case that one or more spaces precede a carriage-return, or a list is terminated with a ')', readline will type '...' and continue reading on the next line,[19] e.g.

```
A B C_↵
...(D E F)
...(X Y Z]
```

and readline returns (A B C (D E F) (X Y Z)).

skread[file;rereadstring][20] is a skip read function. It moves the file pointer for file ahead as if one call to read had been performed, without paying the storage and compute cost to really read in the structure. rereadstring is for the case where the user has already performed some readc's and ratom's before deciding to skip this expression. In this case, rereadstring should be the material already read (as a string), and skread operates as though it

---

19 If the user then types another carriage return, the line will terminate e.g.

```
A B C_↵
...↵
```

and readline returns (A B C)

20 skread was written by J.W. Goodwin.

had seen that material first, thus getting its paren-count, double-quote count, etc. set up properly.

The value of skread is %) if the first thing encountered was a closing paren; %] if the read terminated on an unbalanced %], i.e. one which also would have closed any extant open left parens; otherwise the value of skread is NIL.

14.3 Output Functions

Most of the functions described below have an (optional) argument file which specifies the name of the file on which the operation is to take place. If that argument is NIL, the primary output file will be used.

Note: in all INTERLISP symbolic files, end-of-line is indicated by the characters carriage-return and line-feed in that order. Unless otherwise stated, carriage-return appearing in the description of an output function means carriage-return and line-feed.

prin1[x;file] — prints x on file.

prin2[x;file] — prints x on file with %'s and "'s inserted where required for it to read back in properly by read.

Both prin1 and prin2 print lists as well as atoms and strings; prin1 is usually used only for explicitly printing formatting characters, e.g. (PRIN1 (QUOTE %[)) might be used to print a left square bracket (the % would not be printed by prin1). prin2 is used for printing S-expressions which can then be read back into INTERLISP with read i.e. regular INTERLISP formatting characters in atoms will be preceded by %'s, e.g. the atom '()' is printed as %(%) by prin2. If radix=8, prin2 prints a Q after integers but prin1 does not (but both print the integer in octal).

prin3[x;file] Prints x with %'s and "'s inserted where required for it to read back in properly by uread, i.e. uses separator and break characters specified by setbrk and setsepr to determine when to insert %'s.

print[x;file] Prints the S-expression x using prin2; followed by a carriage-return line-feed. Its value is x.

For all printing functions, pointers other than lists, strings, atoms, or numbers, are printed as #N, where N is the octal representation of the address of the pointer (regardless of radix). Note that this will not read back in correctly, i.e., it will read in as the atom '#N'.

spaces[n;file] Prints n spaces; its value is NIL.

terpri[file] Prints a carriage-return; its value is NIL.

Printlevel

The print functions print, prin1, prin2, and prin3 are all affected by a level parameter set by

printlevel[n] Sets print level to n, value is old setting. Initial value is 1000. printlevel[] gives current setting.

The variable n controls the number of unpaired left parentheses which will be printed. Below that level, all lists will be printed as &.

Suppose x = (A (B C (D (E F) G) H) K). Then if n = 2, print[x] would print (A (B C & H) K), and if n = 3, (A (B C (D & G) H) K), and if n = 0, just &.

If printlevel is *negative*, the action is similar except that a carriage-return is inserted between all occurrences of right parenthesis immediately followed by a left parenthesis.

The printlevel setting can be changed dynamically, even while INTERLISP is printing, by typing control-P followed by a number, i.e. a string of digits, followed by a period or exclamation point.[21] The printlevel will immediately be set to this number.[22] If the print routine is currently deeper than the new level, all unfinished lists above that level will be terminated by "--)". Thus, if a circular or long list of atoms, is being printed out, typing control-P0. will cause the list to be terminated.

If a period is used to terminate the printlevel setting, the printlevel will be returned to its previous setting after this printout. If an exclamation point is used, the change is permanent and the printlevel is not restored (until it is changed again).

Note: printlevel only affects teletype output. Output to all other files acts as though level is infinite.

---

21 As soon as control-P is typed, INTERLISP clears and saves the input buffer, clears the output buffer, rings the bell indicating it has seen the control-P, and then waits for input which is terminated by any non-number. The input buffer is then restored and the program continues. If the input was terminated by other than a period or an exclamation point, it is ignored and printing will continue, except that characters cleared from the output buffer will have been lost.

22 Another way of "turning off" output is to type control-O, which simply clears the output buffer, thereby effectively skipping the next (up to) 64 characters.

14.4 Input/Output Control Functions

clearbuf[file;flg] — Clears the input buffer for *file*. If *file* is T and *flg* is T, contents of INTERLISP's line buffer and the system buffer are saved (and can be obtained via *linbuf* and *sysout* described below). When either control-D, control-E, control-H, control-P, or control-S is typed, INTERLISP automatically does a clearbuf[T;T]. (For control-P and control-S, INTERLISP restores the buffer after the interaction. See Appendix 3.)

linbuf[flg] — if *flg*=T, value is INTERLISP's line buffer (as a string) that was saved at last clearbuf[T;T]. If *flg*=NIL, clears this internal buffer.

sysbuf[flg] — same as *linbuf* for system buffer.

If both the system buffer and INTERLISP's line buffer are empty, the internal buffers associated with *linbuf* and *sysbuf* are not changed by a clearbuf[T;T].

bklinbuf[x] — *x* is a string. *bklinbuf* sets INTERLISP's line buffer to *x*. If greater than 160 characters, first 160 taken.

bksysbuf[x] — *x* is a string. *bksysbuf* sets system buffer to *x*. The effect is the same as though the user typed *x*.

bklinbuf, *bksysbuf*, *linbuf*, and *sysbuf* provide a way of 'undoing' a *clearbuf*. Thus if the user wants to "peek" at various characters in the buffer, he could perform clearbuf[T;T], examine the buffers via *linbuf* and *sysbuf*, and then put them back.

radix[n] Resets output radix[23] to |n| with sign indicator the sign of n. For example, -9 will print as shown with the following radices:

radix	printing
10	-9
-10	68719476727
	i.e. (2↑36-9)
8	-11Q
-8	777777777767Q

Value of radix is its last setting. radix[] gives current setting without changing it. Initial setting is 10.

fltfmt[n] Sets floating format control to n (See TENEX JSYS manual for interpretation of n). fltfmt[T] specifies free format (see Section 3). Value of fltfmt is last setting. fltfmt[] returns current setting without changing it. Initial setting is T.

linelength[n] Sets the length of the print line for all files. Value is the former setting of the line length. Whenever printing an atom would go *beyond* the length of the line, a carriage-return is automatically inserted first. linelength[] returns current setting. Initial setting is 72.

---

23 Currently, there is no input radix.

position[file] — Gives the column number the next character will be read from or printed to, e.g. after a carriage-return, <u>position</u>=0. Note that position[file] is *not* the same as sfptr[file] which gives the position in the *file*, not on the *line*.

<u>Line-buffering and CONTROL</u>

In INTERLISP's normal state, characters typed on the teletype (this section does not apply in any way to input from a file) are transferred to a line buffer. Characters are transmitted from the line buffer to whatever input function initiated the request (i.e., <u>read</u>, <u>uread</u>, <u>ratom</u>, <u>rstring</u>, or <u>readc</u>)[24] *only* when a carriage-return is typed.[25] Until this time, the user can delete characters one at a time from the input buffer by typing control-A. The characters are echoed preceded by a \. Or, the user can delete the entire line buffer back to the last carriage-return by typing control-Q, in which case INTERLISP echoes ##.[26] (If no characters are in the buffer and either control-A or control-Q is typed, INTERLISP echoes ##.)

Note that this line editing is *not* performed by <u>read</u> or <u>ratom</u>, but by INTERLISP, i.e. it does not matter (nor is it necessarily known) which function

---

[24] <u>peekc</u> is an exception; it returns the character immediately.

[25] As mentioned earlier, for calls from <u>read</u> or <u>uread</u>, the characters are also transmitted whenever the parentheses count reaches 0. In this case, if the second argument to <u>read</u> or <u>uread</u> is NIL, INTERLISP also outputs a carriage-return line-feed.

[26] Typing rubout clears the entire input buffer at the time it is *typed*, whereas the action of control-A and control-Q occurs at the time they are read. Rubout can thus be used to clear type-ahead.

will ultimately process the characters, only that they are still in the INTERLISP input buffer. Note also that it is the function that is *currently* requesting input that determines whether parentheses counting is observed, e.g. if the user executes (PROGN (RATOM) (READ)) and types in A (B C D) he will have to type in the carriage-return following the right parenthesis before any action is taken, whereas if he types (PROGN (READ) (READ)) he would not. However, once a carriage-return has been typed, the entire line is 'available' even if not all of it is processed by the function initiating the request for input, i.e. if any characters are 'left over', they will be returned immediately on the next request for input. For example, (PROGN (RATOM) (READC)) followed by A B carriage-return will perform both operations.

Turning-off Line-buffering

The function control is available to defeat this line-buffering. After control[T], characters are returned to the calling function without line-buffering as described below. The function that initiates the request for input determines how the line is treated:

1. read/uread

if the expression being typed is a list, the effect is the same as though control were NIL, i.e. line-buffering until carriage-return or matching parentheses. If the expression being typed is not a list, it is returned as soon as a break or separator character is encountered,[27] e.g. (READ) followed

---

[27] An exception to the above occurs when the break or separator character is a (, ", or [, since returning at this point would leave the line buffer in a "funny" state. Thus if control is T and (READ) is followed by 'ABC(', the ABC will not be read until a carriage-return or matching parentheses is encountered. In this case the user could control-Q the entire line, since all of the characters are still in the buffer.

by ABC space will immediately return ABC. Control-A and control-Q editing are available on those characters still in the buffer. Thus, if a program is performing several reads under control[T], and the user types NOW IS THE TIME followed by control-Q, he will delete only TIME since the rest of the line has already been transmitted to <u>read</u> and processed.

2. <u>ratom</u>

characters are returned as soon as a break or separator character is encountered. Before then, control-A and control-Q may be used as with <u>read</u>, e.g. (RATOM) followed by ABCcontrol-Aspace will return AB. (RATOM) followed by (control-A will return (and type ## indicating that control-A was attempted with nothing in the buffer, since the (is a break character and would therefore already have been read.

3. <u>readc</u>/<u>peekc</u>

the character is returned immediately; no line editing is possible. In particular, (READC) followed by control-A will read the control-A, (READC) followed by % will read the %.

control[u]

<u>u</u>=T	eliminates INTERLISP's normal line-buffering.
u=NIL	restores line-buffering (normal).
u=0	eliminates echo of character being deleted by control-A.
u=1	restores echo (normal).

The value of <u>control</u> when <u>u</u>=T or NIL is its previous line-buffering setting, i.e. T or NIL. When <u>u</u>=0 or 1, its value is its previous echo setting, i.e. 0 or 1.

14.5 Special Functions

sysout[file]

Saves the user's private memory on file. Also saves the stacks, so that if a program performs a sysout, the subsequent sysin will continue from that point, e.g.
(PROGN (SYSOUT (QUOTE FOO)) (PRINT (QUOTE HELLO))) will cause HELLO to be printed after (SYSIN (QUOTE FOO)) The value of sysout is file (full name). A value of NIL indicates the sysout was unsuccessful, i.e., either disk or computer error, or user's directory was full.

Sysout does not save the state of any open files.

Whenever the INTERLISP system is reassembled and/or reloaded, old sysout files are not compatible.

sysin[file]

restores the state of INTERLISP from a sysout file. Value is list[file]. If sysin returns NIL, there was a problem in reading the file. If the file was not found or is incompatible (see note above), generates an error, FILE NOT COMPATIBLE.

Since sysin continues immediately where sysout left off, the only way for a program to determine whether it is just coming back from a sysin or from a sysout is to test the value of sysout.

For example, (COND ((LISTP (SYSOUT (QUOTE FOO))) (PRINT (QUOTE HELLO)))) will cause HELLO to be printed following the sysin, but not when the sysout was performed.

14.6 Symbolic File Input

load[file;ldflg;printflg]

Reads successive S-expressions from file and evaluates each as it is read, until it reads either NIL, or the single atom STOP. Value is file (full name).

If printflg=T, load prints the value of each S-expression; otherwise it does not. ldflg affects the operation of define, defineq, rpaq, and rpaqq. While load is operating, dfnflg (Section 8) is reset to ldflg.[28] Thus, if ldflg=NIL, and a function is redefined, a message is printed and the old definition saved. If ldflg=T, the old definition is simply overwritten. If ldflg=PROP, the function definitions are stored on the property lists under the property EXPR. If ldflg=ALLPROP, not only function definitions but also variables set by rpaqq and rpaq are stored on property lists.[29]

loadfns[fns;file;ldflg][30]

permits selective loading of function definitions. fns is a list of function names, a single function name, or T, meaning all functions (but no

---

[28] Using resetvar (Section 5). dfnflg cannot simply be rebound because it is a GLOBAL variable. See Section 18.

[29] except when the variable has value NOBIND, in which case it is set to the indicated value regardless of dfnflg.

[30] loadfns was written by J.W. Goodwin.

variables, property values, etc.). file can be either a compiled or symbolic file, i.e., any file that can be loaded by load. file is opened and scanned in the manner of load, and every function definition found for a function on fns is loaded.[31] The interpretation of ldflg is the same as for load.

loadfns uses skread (page 14.17) and lcskip (a subfunction of recompile) to skip over undesired material on the file, and so it is very efficient. The value of loadfns is a list of those functions loaded plus a list of those functions not found (if any) headed by the atom NOTFOUND:, e.g., (FOO FIE (NOTFOUND: FUM)).

readfile[file] Reads successive S-expressions from file using read until the single atom STOP is read, or an end of file encountered. Value is a list of these S-expressions.

---

31 If a compiled definition is loaded, so are all compiler generated subfunctions. Note however if fns specifies entries to a block (see Section 18) the user must also specify the block itself.

14.7 Symbolic File Output

writefile[x;file;dateflg] Writes successive S-expressions from x on file. If x is atomic, its value is used. If file is not open, it is opened. If the first expression on x is the type produced by printdate, or if dateflg is T, a new date expression is written. If file is a list, car[file] is used and the file is left opened. Otherwise, when x is finished, a STOP is printed on file and it is closed. Value is file.

pp[x] nlambda, nospread function that performs output[T] and then calls prettyprint: PP FOO is equivalent to PRETTYPRINT((FOO)); PP(FOO FIE) or (PP FOO FIE) is equivalent to PRETTYPRINT((FOO FIE)).
Primary output file is restored after printing.

prettyprint[lst][32] [33] lst is a list of functions (if atomic, its value is used). The definitions of the functions are printed in a pretty format on the primary output file. For example,

```
(FACTORIAL
  [LAMBDA (N)
    (COND
      ((ZEROP N)
        1)
      (T (ITIMES N (FACTORIAL (SUB1 N])
```

---

[32] The prettyprint package was written by W. Teitelman.

[33] prettyprint has a second argument that is T when called from prettydef. In this case, whenever prettyprint starts a new function, it prints (on the teletype) the name of that function if more than 30 seconds (real time) have elapsed since the last time it printed the name of a function.

Note: prettyprint will operate correctly on functions that are broken, broken-in, advised, or have been compiled with their definitions saved on their property lists - it prints the original, pristine definition, but does not change the current state of the function. If prettyprint is given an atom which is not the name of a function, but has a value, it will prettyprint the value.[34] Otherwise, prettyprint will perform spelling correction. If all fails, prettyprint returns (atom NOT PRINTABLE).

Comment Feature

A facility for annotating INTERLISP functions is provided in prettyprint. Any S-expression beginning with * is interpreted as a comment and printed in the right margin. Example:

```
(FACTORIAL
  [LAMBDA (N)                              (* COMPUTES N!)
    (COND
      ((ZEROP N)                           (* 0!=1)
        1)
      (T                                   (* RECURSIVE DEFINITION:
                                           N!=N*N-1!)
         (ITIMES N (FACTORIAL (SUB1 N])
```

These comments actually form a part of the function definition. Accordingly, * is defined as an NLAMBDA NOSPREAD function that returns its argument, i.e. it is equivalent to quote. When running an interpreted function, * is entered the same as any other INTERLISP function. Therefore, comments should only be placed where they will not harm the computation, i.e. where a quoted expression could be placed. For example, writing (ITIMES N (FACTORIAL (SUB1 N)) (* RECURSIVE DEFINITION)) in the above function would cause an error when ITIMES attempted to multiply N, N-1!, and RECURSIVE.

---

[34] except when prettyprint is called from prettydef.

For compilation purposes, * is defined as a macro which compiles into no instructions. Thus, if you compile a function with comments, and load the compiled definition into another system, the extra atom and list structures storage required by the comments will be eliminated. This is the way the comment feature is intended to be used. For more options, see end of this section.

Comments are designed mainly for documenting *listings*. Thus when prettyprinting to the teletype, comments are suppressed and printed as the string **COMMENT**.[35]

Prettydef

prettydef[prettyfns;prettyfile;prettycoms][36] Used to make symbolic files that are suitable for loading which contain function definitions, variable settings, property lists, et al, in a prettyprint format.

The arguments are interpreted as follows:

prettyfns — Is a list of function names. The functions on the list are prettyprinted surrounded by a (DEFINEQ ...) so that they can be

---

[35] The value of **comment**flg determines the action. If **comment**flg is NIL, the comment is printed. Otherwise, the value of **comment**flg is printed. **comment**flg is initially set to " **COMMENT** ". The function pp* is provided to prettyprint functions, including their comments, to the teletype. pp* operates exactly like pp except it first sets **comment**flg to NIL.

[36] prettydef actually has two additional arguments for use by the system.

loaded with load. If prettyfns is atomic (the preferred usage), its top level value is used as the list of function names, and an rpaqq [37] will also be written which will set that atom to the list of functions when the file is loaded. A print expression will also be written which informs the user of the named atom or list of functions when the file is subsequently loaded.

prettyfile — is the name of the file on which the output is to be written.

The following options exist:

prettyfile=NIL

The primary output file is used.

prettyfile atomic

The file is opened if not already open, and becomes primary output file. File is closed at end of prettydef and primary output file is restored.

prettyfile a list

Car of the list is assumed to be the file name, and is opened if not already open. The file is left open at end of prettydef.

---

[37] rpaqq and rpaq are like setqq and setq, except they set the top level value. See Section 5.

prettycoms — Is a list of commands interpreted as described below. If prettycoms is atomic (the preferred usage), its top level value is used and an rpaqq is written which will set that atom to the list of commands when the file is subsequently loaded, exactly as with prettyfns.

These commands are used to save on the output file top level bindings of variables, property lists of atoms, miscellaneous INTERLISP forms to be evaluated upon loading, arrays, and advised functions. It also provides for evaluation of forms at ouput time.

The interpretation of each command in the command list is as follows:

1. if atomic, an rpaqq is written which will restore the top level value of this atom when the file is loaded.

2. (PROP propname $atom_1$ ... $atom_n$) an appropriate deflist will be written which will restore the value of propname for each $atom_i$ when the file is loaded. If propname=ALL, the values of all user properties (on the property list of each $atom_i$) are saved.[38] If propname is a list, deflist's will be written for each property on that list.

3. (ARRAY $atom_1$ ... $atom_n$), each atom following ARRAY should have an array as its value. An appropriate expression will be written which will set the atom to an array of exactly the same size, type, and contents upon loading.

---

[38] sysprops is a list of properties used by system functions. Only properties *not* on that list are dumped when the ALL option is used.

4. (P ...), each S-expression following P will be printed on the output file, and consequently evaluated when the file is loaded.

5. (E ...), each form following E will be evaluated at output time, i.e., when prettydef reaches this command.

6. (FNS fn_1 ...fn_m), a defineq is written with the definitions of fn_1 ... fn_m exactly as though (fn_1 ...fn_m) where the first argument to prettydef. For example, suppose the user wanted to set some variables or perform some computations in a file *before* defining functions, he would then write the definitions using the FNS command instead of the first argument to prettydef.

7. (VARS var_1 ... var_n), for each var_i, an expression will be written which will set its top level value when the file is loaded. If var_i is atomic, var_i will be set to the top-level value it had at the time the file was prettydefed, i.e. (RPAQQ var_i top-level-value) is written. If var_i is non-atomic, it is interpreted as (var form). e.g. (FOO (APPEND FIE FUM)) or (FOO (QUOTE (FOO1 FOO2 FOO3))). In this case the expression (RPAQ var form) is written.

8. (ADVISE fn_1 ... fn_m), for each fn_n, an appropriate expression will be written which will reinstate the function to its advised state when the file is loaded.

9. (ADVICE fn_1 ... fn_m,), for each fn_i, will write a deflist which will put the advice back on the property list of the function. The user can then use readvise to reactivate the advice. See Section 19.

10. (BLOCKS $block_1$ ... $block_n$) for each $block_i$, a declare expression will be written which the block compile functions interpret as block declarations. See Section 18.

11. (COMS com_1 ... com_n), each of the commands com_1 ... com_n will be interpreted as a prettydef command.

12. (ADDVARS (var_1 . lst_1) ... (var_n . lst_n)) For each var_i, the effect is the same as (RPAQ var_i (UNION lst_i var_i)), i.e. each element of lst_i not a member of var_i (at load time) is added to it. var_i can initially be NOBIND, in which case it is first set to NIL.

13. (USERMACROS $atom_1$ ... $atom_n$), each $atom_i$ is the name of a user edit macro. USERMACROS writes expressions for adding the definitions to usermacros and the names to the appropriate spelling lists. (USERMACROS) will save all user edit macros.

14. (IFPROP propname $atom_1$ ... $atom_n$) same as PROP command, except that only non-NIL property values are saved. For example, if FOO1 has property PROP1 and PROP2, FOO2 has PROP3, and FOO3 has property PROP1 and PROP3, (IFPROP (PROP1 PROP2 PROP3) FOO1 FOO2 FOO3) will save only those 5 property values.

15. (COMPROP propname atom, ... atomn) same as PROP command, except that the corresponding deflist expression will also be evaluated when the file is compiled. Useful for outputting MACROs.

16. (COMPROP* propname atom, ... atomn), same as COMPROP except that the corresponding deflist expressions are not copied to the compiled file by tcompl, bcompl, recompile, or brecompile.

17. (PD ...), like P except that the corresponding S-expressions are also printed as DECLARE expressions, and thus will be evaluated when the file is compiled. In other words, (PD (DEFLIST (QUOTE --) (QUOTE propname)) is essentially equivalent to (COMPROP propname --).

In each of the commands described above, if the atom * follows the command type, the form following the *, i.e., caddr of the command, is evaluated and its value used in executing the command, e.g., (FNS * (APPEND FNS1 FNS2)).[39] Note that (COMS * form) provides a way of *computing* what should be done by prettydef.

New prettydef commands can be defined via prettymacros (see page 14.40). If prettydef is given a command not one of the above, and not defined on prettymacros, it attempts spelling correction[40] using prettycomsplst as a spelling list. If successful, the corrected version of prettycoms is written (again) on the output file.[41] If unsuccessful, prettydef generates an error, BAD PRETTYCOM.

---

[39] Except for the PROP, IFPROP, COMPROP, and COMPROP* commands, in which case the * must follow the property name, e.g., (PROP MACRO * FOOMACROS).

[40] unless dwimflg=NIL. See Section 17.

[41] since at this point, the uncorrected prettycoms would already have been printed on the output file. When the file is loaded, this will result in prettycoms being reset, and a message printed, e.g. (FOOVARS RESET). The value of FOOVARS would then be the corrected version.

Example:

```
←SET(FOOFNS (FOO1 FOO2 FOO3))
←SET(FOOVARS(FIE (PROP MACRO FOO1 FOO2) (P (MOVD (QUOTE FOO1)
  (QUOTE FIE1]
←PRETTYDEF(FOOFNS FOO FOOVARS)
```

would create a file FOO containing

1. A message which prints the time and date the file was made (done automatically)
2. DEFINEQ followed by the definitions of FOO1, FOO2, and FOO3
3. (PRINT (QUOTE FOOFNS) T)
4. (RPAQQ FOOFNS (FOO1 FOO3 FOO3))
5. (PRINT (QUOTE FOOVARS) T)
6. (RPAQQ FOOVARS (FIE ...)
7. (RPAQQ FIE value of *fie*)
8. (DEFLIST (QUOTE ((FOO1 propvalue) (FOO2 propvalue))) (QUOTE MACRO))
9. (MOVD (QUOTE FOO1) (QUOTE FIE1))
10. STOP

printfns[x] — *x* is a list of functions. *printfns* prints *defineq* and prettyprints the functions. Used by *prettydef*, i.e. command (FNS * FOO) is equivalent to command (E (PRINTFNS FOO)).

printdate[file;changes] — prints the expression at beginning of *prettydefed* files that upon loading types the time and date the file was made, and stores this time and date on the property list of *file* under the property FILEDATE. *changes* is for use by the file package.

tab[pos;minspaces;file] — performs appropriate number of spaces to move to

position pos. minspaces indicates the minimum number of spaces to be printed by tab, i.e., it is intended to be small number (if NIL, 1 is used). Thus, if position + minspaces is greater than pos, tab does a terpri and then spaces[pos].

endfile[file] — Prints STOP on file and closes it.

printdef[expr;left;def] — prints the expression expr on the primary output file in a pretty format. left is the left hand margin (linelength determines the right hand margin). 2 is used if left=NIL.

def=T means expr is a function definition, or a piece of one, i.e. prettyprint is essentially printdef[getd[fn];NIL;T]. If def=NIL, no special action will be taken for LAMBDA's, PROG's, COND's, comments, CLISP, etc. def is NIL when prettydef calls prettyprint to print variables and property lists, and when printdef is called from the editor via the command PPV.

Special Prettyprint Controls

All variables described below, i.e., #rpars, firstcol, et al, are globalvars, see Section 18. Therefore, if they are to be changed, they must be *reset*, not rebound.

#rpars — controls the number of right parentheses necessary for square bracketing to occur. If #rpars=NIL, no brackets are used. #rpars is initialized to 4.

linelength[n] — determines the position of the right margin for prettyprint.

firstcol — is the starting column for comments. Initial setting is 48. Comments run between firstcol and linelength. If a word in a comment ends with a '.' and is not on the list abbrevlst, and the position is greater than halfway between firstcol and linelength, the next word in the comment begins on a new line. Also, if a list is encountered in a comment, and the position is greater than halfway, the list begins on a new line.

prettylcom — If a comment is bigger (using count) than prettylcom in size, it is printed starting at column 10, instead of firstcol. prettylcom is initialized to 14 (arrived at empirically).

widepaper[flg] — widepaper[T] sets linelength to 120, firstcol to 80 and prettylcom to 28. This is a useful setting for prettyprinting files to be listed on wide paper. widepaper[] restores these parameters to their initial values. The value of widepaper is its previous setting.

commentflg — If car of an expression is eq to commentflg, the expression is treated as a comment. commentflg is initialized to *.

prettyflg — If prettyflg is NIL, printdef uses prin2 instead

of prettyprinting. This is useful for producing a fast symbolic dump (e.g. when TENEX is very slow.) Note that the file loads the same as if it were prettyprinted. prettyflg is initially set to T.

clispifyprettyflg — if T, causes prettyprint to clispify each function definition before printing. See Section 23. clispifyprettyflg is initially NIL.

prettymacros — Is an assoc-type list for defining substitution macros for prettydef. If (FOO (X Y) . coms) appears on prettymacros, then (FOO A B) appearing in the third argument to prettydef will cause A to be substituted for X and B for Y throughout coms (i.e., cddr of the macro), and then coms treated as a list of commands for prettydef.

(* E x) — A comment of this form causes x to be evaluated at prettyprint time, e.g., (* E (RADIX 8)) as a comment in a function containing octal numbers can be used to change the radix to produce more readable printout. The comment is also printed.

Converting Comments to Lower Case

This section is for users operating on terminals without lower case who nevertheless would like their comments to be converted to lower case for more readable line-printer listings. Users with lower-case terminals can skip to the File Package sections (as they can type comments directly in lower case).

%% If the second atom in a comment is %%, the text of the comment is converted to lower case so that it looks like English instead of LISP (see next page).

The output on the next page illustrates the result of a lower casing operation. Before this function was <u>prettydefed</u>, all comments consisted of upper case atoms, e.g., the first comment was (* %% INTERPRETS A SINGLE COMMAND). Note that comments are converted *only* when they are actually written to a file by <u>prettydef</u>.

The algorithm for conversion to lower case is the following: If the first character in an atom is ↑, do not change the atom (but remove the ↑). If the first character is %, convert the atom to lower case.[42] If the atom[43] is an INTERLISP word,[44] do not change it. Otherwise, convert the atom to lower case. Conversion only affects the upper case alphabet, i.e., atoms already converted to lower case are not changed if the comment is converted again. When converting, the first character in the comment and the first character following each period are left capitalized. After conversion, the comment is physically modified to be the lower case text minus the %% flag, so that conversion is thus only performed once (unless the user edits the comment inserting additional upper case text and another %% flag).

---

[42] User must type %% as % is the escape character.

[43] minus any trailing punctuation marks.

[44] i.e., is a bound or free variable for the function containing the comment, or has a top level value, or is a defined function, or has a non-NIL property list.

```
(BREAKCOM
  [LAMBDA (BRKCOM BRKFLG)                    (* Interprets a
                                             single command.)
    (PROG (BRKZ)
      TOP (SELECTQ
            BRKCOM
            [↑ (RETEVAL (QUOTE BREAK1)
                        (QUOTE (ERROR]]
            (GO                              (* Evaluate BRKEXP
                                             unless already evaluated,
                                             print value, and exit.)
                (BREAKCOM1 BRKEXP BRKCOM NIL BRKVALUE)
                (BREAKEXIT))
            (OK                              (* Evaluate BRKEXP,
                                             unless already evaluated,
                                             do NOT print value,
                                             and exit.)
                (BREAKCOM1 BRKEXP BRKCOM BRKVALUE BRKVALUE)
                (BREAKEXIT T))
            (↑WGO                            (* Same as GO except
                                             never saves evaluation
                                             on history.)
                (BREAKCOM1 BRKEXP BRKCOM T BRKVALUE)
                (BREAKEXIT))
            (RETURN

       (* User will type in expression to be evaluated and
       returned as value of BREAK. Otherwise same as GO.)


                  (BREAKCOM1 [SETQ BRKZ (COND
                                   (BRKCOMS (CAR BRKCOMS))
                                   (T (LISPXREAD T]
                             (QUOTE RETURN)
                             NIL NIL (LIST (QUOTE RETURN)
                                           BRKZ))
                  (BREAKEXIT))
       (EVAL                                 (* Evaluate BRKEXP but
                                             do not exit from BREAK.)
            (BREAKCOM1 BRKEXP BRKCOM)
            (COND
              (BRKFLG (BREAK2)
                      (PRIN1 BRKFN T)
                      (PRIN1 (QUOTE " EVALUATED
")
                             T)))
              (SETQ !VALUE (CAR BRKVALUE))
                                             (* For user's benefit.)
              )
```

lcaselst

Words on lcaselst will always be converted to lower case. lcaselst is initialized to contain words which are INTERLISP functions but also appear frequently in comments as English words. e.g. AND, EVERY, GET, GO, LAST, LENGTH, LIST, etc. Thus, in the example on the previous page, not was written as ↑NOT, and GO as ↑GO in order that they might be left in upper case.

ucaselst

words on ucaselst (that do not appear on lcaselst) will be left in upper case. ucaselst is initialized to NIL.

abbrevlst

abbrevlst is used to distinguish between abbreviations and words that end in periods. Normally, words that end in periods and occur more than halfway to the right margin cause carriage returns. Furthermore, during conversion to lowercase, words ending in periods, except for those on abbrevlst, cause the first character in the *next* word to be capitalized. abbrevlst is initialized to the upper and lower case forms of ETC. I.E. and E.G.

l-case[x;flg]

value is lower case version of x. If flg is T, the first letter is capitalized, e.g. l-case[FOO;T] = Foo, l-case[FOO] = foo. If x is a string, the value of l-case is also a string, e.g. l-case["FILE NOT FOUND";T] = "File not found".

u-case[x]

Similar to l-case

14.8 File Package[45]

This section describes a set of functions and conventions for facilitating the bookkeeping involved with working in a large system consisting of many symbolic files and their compiled counterparts. The file package keeps track of which files have been in some way modified and need to be dumped, which files have been dumped, but still need to be listed and/or recompiled. The functions described below comprise a coherent package for eliminating this burden from the user. They require that for each file, the first argument to prettydef, (if any), be an atom of the form fileFNS, and the third argument, (if any), be fileVARS where file is the name of the file, e.g. prettydef[FOOFNS;FOO;FOOVARS].[46]

The functions load, editf, editv, tcompl, recompile, bcompl, brecompile, and DWIM interact with the functions and global variables in the file package as follows. Whenever load is called, its argument is added to the list filelst, and the property FILE, value (fileFNS fileVARS), is added to the property list of the file name.[47] This property value is used to determine whether or not the file has been modified since the last time it was loaded or dumped. Whenever the user calls editf and changes a function, filelst is searched to find the

---

[45] The file package was written by W. Teitelman. It can be disabled by setting filepkgflg to NIL.

[46] file can contain a suffix and/or version number, e.g. PRETTYDEF(FOOFNS FOO.TEM;3 FOOVARS) is acceptable. The essential point is that the FNS and VARS be computable from the name of the file.

[47] The name added to filelst has the version number and directory field removed, if any. fileFNS and fileVARS are constructed using only the name field, i.e., if the user performs load[<TEITELMAN>FOO.TEM;2], FOO.TEM is added to filelst, and (FOOFNS FOOVARS) put on the property list of FOO.TEM. If the file was originally made under a different name, fileFNS and fileVARS are computed from the original name (which is obtained from the expression that printdate puts at the beginning of the file).

files[48] containing this function, i.e. the files for which the function was either a member of fileFNS, or appeared in a FNS command on fileVARS. When (if) such files are found, the name of the function is added, using /nconc, to the value of the property FILE for each file. Thus if the user loads the file FOO containing definitions for FOO1, FOO2, and FOO3, and then edits FOO2, getp[FOO;FILE] will be (FOOFNS FOOVARS FOO2) following the edit. A similar update takes place for calls to editv.

Whenever the user dumps a file using makefile (described below), the file is added to filelst (if not already there) and its FILE property is reinitialized to (fileFNS fileVARS), indicating that the file is up to date. In addition, the file is added to the list notlistedfiles and notcompiledfiles. Whenever the user lists a file using listfiles, it is removed from notlistedfiles. Similarly, whenever a file is compiled by tcompl, recompile, bcompl, or brecompile, the file is removed from notcompiledfiles. Thus at each point, the state of all files can be determined. This information is available to the user via the function files?. Similarly, the user can see whether and how each particular file has been modified, dump all files that have been modified, list all files that have been dumped but not listed, recompile all files that have been dumped but not recompiled, or any combination of any or all of the above by using one of the function described below.

---

48 If the user has many files, with complex prettydef commands, this procedure could be time consuming. Therefore, in the interests of efficiency, what really happens is the function name is simply consed onto the front of the list changedfnslst, and variable names are consed onto changedvarslst. Whenever the user calls files?, cleanup, makefiles, or any other operation that actually looks at the FILE property, the function updatefiles is called which scans changedfnslst and changedvarslst and moves the function/variable names to the appropriate property lists. The user can explicitly perform this updating process by calling updatefiles. prettytypelst, page 14.50, provides a way of informing updatefiles about user-defined types in addition to functions or variables.

makefile[file;options] adds file to filelst if not already there. Calls prettydef[fileFNS;file;fileVARS;NIL;changes],[49] and then adds file to notlistedfiles, notcompiledfiles.[50] options is a list of options or a single option interpreted as follows:

FAST perform prettydef with prettyflg=NIL

RC call recompile after prettydef or brecompile if there are any block declarations specified in fileVARS.[51]

C calls tcompl after prettydef or bcompl if there are any block declarations specified in fileVARS.

CLISPIFY perform prettydef with clispifyprettyflg=T, causing clispify (see Section 23) to be called on each function definition before it is prettyprinted.[52]

NOCLISP performs prettydef with prettytranflg=T, causing CLISP translations to be printed, if any, in place of the corresponding CLISP expression, e.g. iterative statement.

---

[49] fileFNS and fileVARS are constructed from the name field only, e.g. makefile[FOO.TEM] will work. changes is cddr of the FILE property, i.e. those items that have been changed since the last makefile. prettydef merges those changes with those handled in previous calls to makefile, and stores the result on the property FILEDATE in the form (date1 date2 changes), where date1 is the date of the file that was originally loaded, date2 the date of the latest version (i.e. this one), and changes (the union of) all items that differ in the two files. printdate also includes this information in the expression printed at the beginning of the file.

[50] except for files that do not contain any function definitions or those that have on their property list the property FILETYPE with value DON'TCOMPILE. Such files are not compiled even when options specifies C or RC, nor are they added to notcompiledfiles.

[51] Including any generated via the COMS command or via a prettymacro.

[52] Another way to accomplish this is to put on the property list of the file under the property FILETYPE the value CLISP. In this case, the compiler will also know to dwimify the functions before compiling.

LIST calls listfiles on file.

For the three compile options, if F or ST is the *next* option, it will be given to the compiler as the answer to the compiler's question LISTING?, e.g. makefile[FOO;(C F.LIST)] will dump FOO, then tcompl or bcompl it without redefining any functions, and finally list the file.

The user can indicate that file must be block compiled with other files by putting a list of those files on the property list of each file under the property FILEGROUP;. For example, EDIT and WEDIT are one such group, DWIM, FIX, CLISP, and DWIMIFY another. If file has a FILEGROUP; property, the compiler will not be called until all files on this property have been dumped that need to be. In the case of recompiling, brecompile will be called with coreflg=T only if all of the files in the group are currently in core.

makefiles[options;files] For each file on files that has been changed,[53] performs makefile[file;options], If files = NIL, filelst is used, e.g. makefiles[LIST] will make and list all files.[54] Value is a list of all files that are made.

listfiles[files] nlambda, nospread function. Uses bksysbuf to load system buffer appropriately to list each file on files, (if NIL, notlistedfiles is used) followed

---

[53] except if the file is a compiled file, a message is printed and the makefile not performed. For example, if the user loads FOO.COM and then edits FOOVARS, when makefiles is called, the message "FOO.COM IS A COMPILED FILE AND CANNOT BE DUMPED." is printed.

[54] In this case, if any functions have been defined or changed that are not contained in one of the files on filelst, a message is printed alerting the user.

by a QUIT command, then calls a lower EXEC via subsys (section 21). The EXEC will then read from the system buffer, list the files, and QUIT back to the program.

Each file listed is removed from notlistedfiles if the listing is completed, e.g. if the user control-C's to stop the listing and QUITS.

compilefiles[files] — nlambda, nospread function. Executes the RC option of makefile for each member of files. (If files=NIL, notcompiledfiles is used.)

files?[] — Prints on terminal the names of those files that have been modified but not dumped, dumped but not listed, dumped but not compiled, plus the names of those functions (if any) that are not contained in any file.

cleanup[files] — nlambda, nospread. Dumps, lists, and recompiles (or brecompiles) any and all files on files requiring the corresponding operation. If files = NIL, filelst is used. Value is NIL.

Note: if both a compiled and symbolic version of the same[55] file appear on filelst, the compiled file is ignored by makefiles, files?, and cleanup.

whereis[x] — x is either the name of a function or variable.

---

[55] i.e. the compiled file has a COM suffix and its fns and vars are the same as those of another (symbolic) file on filelst.

whereis sweeps through all the files on sysfiles and filelst looking for files that define or set x, and prints the names of all such files found. whereis knows about and expands all prettydef commands and prettymacros.

Note that whereis requires that the fileFNS and fileVARS of the files be available. However, the system fileFNS and fileVARS are clobbered to save space. To get them back, load the file <LISP>FNS/VARS.

filefnslst[file]

returns a list of the functions in file, i.e. specified by fileFNS and fileVARS. filefnslst knows about prettymacros.

newfile2[name;coms;type]

coms is a list of prettydef commands, type is usually FNS or VARS but may be BLOCKS, ARRAYS, etc. or the name of any other prettydef command. If name=NIL, newfile2 returns a list of all elements of type type. (filefnslst and bcompl and brecompile use this option.)

If name=T, newfile2 returns T if there are *any* elements of type type. (makefile uses this option to determine whether the file contains any FNS, and therefore should be compiled, and if so, whether it contains any BLOCKS, to determine whether to call bcompl/brecompile or tcompl/recompile.)

Otherwise, newfile2 returns T if name is "contained" in coms. (whereis uses newfile2 in this way.)

*　　　*　　　*

If the user often employs prettymacros, their expansion by the various parts of the system that need to interrogate files can result in a large number of conses and garbage collections. If the user could inform the file package as to what his various prettymacros actually produce, this expansion would not be necessary. For example, the user may have a macro called GRAMMARS which dumps various property list but no functions. Thus, the file package could ignore this command when seeking information about FNS. The user can supply this information by putting on the property list of the prettymacro, e.g. GRAMMARS, under the property PRETTYTYPE,[56] a function (or LAMBDA expression) of two arguments, com and type, where com is a prettydef command, and type is FNS, VARS, BLOCKS, etc. The result of applying the function to these arguments should be a list of those elements of type type contained in com. For example, the function corresponding to GRAMMARS might be (LAMBDA(COM TYPE)(AND (EQ (CAR COM) TYPE)(EQ TYPE(QUOTE GRAMMARS))(CDR COM)).[57]

Currently, the file package knows about two "types": functions and variables. As described in footnote on page 14.45, whenver a function or variable is changed, it is added to changedfnslst or changedvarslst respectively. Updatefiles operates by mapping down filelst and using newfile2 to determine if the corresponding file contains any of the functions on changedfnslst or changedvarslst. The user can tell the file package about other types by adding appropriate entries to prettytypelst. Each element of prettytypelst is a list of the form (name-of-changedlist type string), where string is optional. For

---

[56] If nothing appears on property PRETTYTYPE, the command is expanded as before.

[57] Note that since the function is given the entire command as an argument, the same function could be used for several different types.

example, prettytypelst is initially ((CHANGEDFNSLST FNS "functions") (CHANGEDVARSLST VARS)).[58] If the user adds (CHANGEDGRAMLST GRAMMARS) to prettytypelst, then updatefiles will know to move elements on changedgramlst to the FILE property for the files that contain them.[59]

---

[58] If string is supplied, files? will inform the user if any elements remain on the changed list after updatefiles has completed. Similarly, makefiles will warn the user that some elements of this type are not going to be dumped in the event that it could not find the file to which they belonged.

[59] It is the user's responsibility to see that elements are added to the changed list in the first place.

Index for Section 14

Page
Numbers

Page Numbers

Page Numbers

Page Numbers

SECTION 15

DEBUGGING - THE BREAK PACKAGE[1]

15.1 Debugging Facilities

Debugging a collection of LISP functions involves isolating problems within particular functions and/or determining when and where incorrect data are being generated and transmitted. In the INTERLISP system, there are three facilities which allow the user to (temporarily) modify selected function definitions so that he can follow the flow of control in his programs, and obtain this debugging information. These three facilities together are called the break package. All three redefine functions in terms of a system function, breaki described below.

Break modifies the definition of its argument, a function fn, so that if a break condition (defined by the user) is satisfied, the process is halted temporarily on a call to fn. The user can then interrogate the state of the machine, perform any computation, and continue or return from the call.

Trace modifies a definition of a function fn so that whenever fn is called, its arguments (or some other values specified by the user) are printed. When the value of fn is computed it is printed also. (trace is a special case of break).

---

[1] The break package was written by W. Teitelman.

Breakin allows the user to insert a breakpoint *inside* an expression defining a function. When the breakpoint is reached and if a break condition (defined by the user) is satisfied, a temporary halt occurs and the user can again investigate the state of the computation.

The following two examples illustrate these facilities. In the first example, the user traces the function factorial. trace redefines factorial so that it calls break1 in such a way that it prints some information, in this case the arguments and value of factorial, and then goes on with the computation. When an error occurs on the fifth recursion, break1 reverts to interactive mode, and a full break occurs. The situation is then the same as though the user had originally performed BREAK(FACTORIAL) instead of TRACE(FACTORIAL), and the user can evaluate various INTERLISP forms and direct the course of the computation. In this case, the user examines the variable n, and instructs break1 to return 1 as the value of this cell to factorial. The rest of the tracing proceeds without incident. The user would then presumably edit factorial to change L to 1.

In the second example, the user has constructed a non-recursive definition of factorial. He uses breakin to insert a call to break1 just after the PROG label LOOP. This break is to occur only on the last two iterations, i.e., when n is less than 2. When the break occurs, the user looks at the value of n. mistakenly typing NN. However, the break is maintained and no damage is done. After examining n and m the user allows the computation to continue by typing OK. A second break occurs after the next iteration, this time with N=0. When this break is released, the function factorial returns its value of 120.

```
←PP FACTORIAL

(FACTORIAL
  [LAMBDA (N)
    (COND
      ((ZEROP N
         L)
      (T (ITIMES N (FACTORIAL (SUB1 N])
FACTORIAL
←TRACE(FACTORIAL)
(FACTORIAL)
←FACTORIAL(4)

FACTORIAL:
N = 4

   FACTORIAL:
   N = 3

      FACTORIAL:
      N = 2

         FACTORIAL:
         N = 1

            FACTORIAL:
            N = 0

U.B.A.
L
(FACTORIAL BROKEN)
:N
0
:RETURN 1
            FACTORIAL = 1
         FACTORIAL = 1
      FACTORIAL = 2
   FACTORIAL = 6
FACTORIAL = 24
24
←
```

```
←PP FACTORIAL

(FACTORIAL
  [LAMBDA (N)
    (PROG ((M 1))
      LOOP(COND
             ((ZEROP N)
               (RETURN M)))
          (SETQ M (ITIMES M N))
          (SETQ N (SUB1 N))
          (GO LOOP])
FACTORIAL
←BREAKIN(FACTORIAL (AFTER LOOP) (ILESSP N 2]
SEARCHING...
FACTORIAL
←FACTORIAL(5)

((FACTORIAL) BROKEN)
:NN
U.B.A.
NN
(FACTORIAL BROKEN AFTER LOOP)
:N
1
:M
120
:OK
(FACTORIAL)

((FACTORIAL) BROKEN)
:N
0
:OK
(FACTORIAL)
120
←
```

15.2 Break1

The basic function of the break package is break1. Whenever INTERLISP types a message of the form (- BROKEN) followed by ':' the user is then 'talking to' break1, and we say he is 'in a break.' break1 allows the user to interrogate the state of the world and affect the course of the computation. It uses the prompt character ':' to indicate it is ready to accept input(s) for evaluation, in the same way as evalqt uses '←'. The user may type in an expression for evaluation as with evalqt, and the value will be printed out, followed by another :. Or the user can type in one of the commands specifically recognized by break1 described below.

Since breaki puts all of the power of INTERLISP at the user's command, he can do anything he can do at evalqt. For example, he can insert new breaks on subordinate functions simply by typing:

```
(BREAK fn1 fn2 ...)
```

or he can remove old breaks and traces if too much information is being supplied:

```
(UNBREAK fn3 fn4 ...)
```

He can edit functions, including the one currently broken:

```
EDITF(fn)
```

For example, the user might evaluate an expression, see that the value was incorrect, call the editor, change the function, and evaluate the expression again, all without leaving the break.

Similarly, the user can prettyprint functions, define new functions or redefine old ones, load a file, compile functions, time a computation, etc. In short, anything that he can do at the top level can be done while inside of the break. In addition the user can examine the pushdown list, via the functions described in Section 12, and even force a return back to some higher function via the function retfrom or reteval.

It is important to emphasize that once a break occurs, the user is in complete control of the flow of the computation, and the computation will not proceed without specific instruction from him. If the user types in an expression whose evaluation causes an error, the break is maintained. Similarly if the

user aborts a computation[2] initiated from within the break, the break is maintained. Only if the user gives one of the commands that exits from the break, or evaluates a form which does a retfrom or reteval back out of break1, will the computation continue.[3]

Note that break1 is just another INTERLISP function, not a special system feature like the interpreter or the garbage collector. It has arguments which are explained later, and returns a value, the same as cons or cond or prog or any other function. The value returned by break1 is called 'the value of the break.' The user can specify this value explicitly by using the RETURN command described below. But in most cases, the value of a is given implicitly, via a GO or OK command, and is the result of evaluating 'the break expression,' brkexp, which is one of the arguments to break1.

The break expression is an expression equivalent to the computation that would have taken place had no break occurred. For example, if the user breaks on the function FOO, the break expression is the body of the definition of FOO. When the user types OK or GO, the body of FOO is evaluated, and its value returned as the value of the break, i.e. to whatever function called FOO. The effect is the same as though no break had occurred. In other words, one can think of break1 as a fancy eval, which permits interaction before and after evaluation. The break expression then corresponds to the argument to eval.

---

2 By typing control-E, see Section 16.

3 Except that break1 does not 'turn off' control-D, i.e. a control-D will force an immediate return back to the top level.

Break Commands

GO

Releases the break and allows the computation to proceed. breaki evaluates brkexp, its first argument, prints the value of the break. brkexp is set up by the function that created the call to breaki. For break or trace, brkexp is equivalent to the body of the definition of the broken function. For breakin, using BEFORE or AFTER, brkexp is NIL. For breakin AROUND, brkexp is the indicated expression. See breakin, page 15.19.

OK

Same as GO except the value of brkexp is not printed.

EVAL

Same as GO or OK except that the break is maintained after the evaluation. The user can then interrogate the value of the break which is bound on the variable !value, and continue with the break. Typing GO or OK following EVAL will not cause reevaluation but another EVAL will. EVAL is a useful command when the user is not sure whether or not the break will produce the correct value and wishes to be able to do something about it if it is wrong.

RETURN form
or
RETURN fn[args]

The value of the indicated computation is returned as the value of the break.
For example, one might use the EVAL command and follow this with RETURN (REVERSE !VALUE).

↑

Calls error! and aborts the break. i.e. makes it "go away' without returning a value. This is a useful way to unwind to a higher level break. All other errors, including those encountered while executing the GO, OK, EVAL, and RETURN commands, maintain the break.

!EVAL

function is first unbroken, then evaluated, and then rebroken. Very useful for dealing with recursive functions.

!OK

Function is first unbroken, evaluated, rebroken, and then exited, i.e. !OK is equivalent to !EVAL followed by OK.

!GO

Function is first unbroken, evaluated, rebroken, and exited with value typed, i.e., !EVAL followed by GO.

UB

unbreaks brkfn, e.g.

```
(FOO BROKEN)
:UB
FOO
:
```

and FOO is now unbroken

@

resets the variable lastpos, which establishes a context for the commands ?=, ARGS, BT, BTV, BTV*, and EDIT, and IN? described below. lastpos is the position of a function call on the push-down stack. It is initialized to the function just before the call to break1, i.e. stknth[-1;BREAK1]

@ treats the rest of the teletype line as its argument(s). It first resets lastpos to stknth[-1;BREAK1] and then for each atom on the line, @ searches backward, for a call to that atom. The following atoms are treated specially:

@ do not reset lastpos to stknth[-1;BREAK1] but leave it as it was, and continue searching from that point.

numbers if negative, move lastpos back that number of calls, if positive, forward, i.e. reset lastpos to stknth[n;lastpos]

← search forward for next atom

/ the next atom is a number and can be used to specify more than one call e.g. @ FOO / 3 is equivalent to @ FOO FOO FOO

Example:

if the push-down stack looks like

```
BREAK1    (13)
FOO       (12)
SETQ      (11)
COND      (10)
PROG       (9)
FIE        (8)
COND       (7)
FIE        (6)
COND       (5)
FIE        (4)
COND       (3)
PROG       (2)
FUM        (1)
```

then @ FIE COND will set lastpos to the position corresponding to (7); @ @ COND will then set lastpos to (5); @ FUM ← FIE to (4); and @ FIE / 3 -1 to (3).

If @ cannot successfully complete a search, it

types (fn NOT FOUND), where fn is the name of the function for which it was searching.

When @ finishes, it types the name of the function at lastpos, i.e. stkname[lastpos]

@ can be used on brkcoms. In this case, the next command on brkcoms is treated the same as the rest of the teletype line.

?=

This is a multi-purpose command. Its most common use is to interrogate the value(s) of the arguments of the broken function, e.g. if FOO has three arguments (X Y Z), then typing ?= to a break on FOO, will produce:

```
:?=
X =      value of X
Y =      value of Y
Z =      value of Z
:
```

?= operates on the rest of the teletype line as its arguments. If the line is empty, as in the above case, it prints all of the arguments. If the user types ?= X (CAR Y), he will see the value of X, and the value of (CAR Y). The difference between using ?= and typing X and (CAR Y) directly to breaki is that ?= evaluates its inputs as of lastpos, i.e. it uses stkeval. This provides a way of examing variables or performing computations *as of a particular point on the stack*. For example, @ FOO / 2 followed by ?= X will allow the user to examine the value of X in the previous call to FOO, etc.

?= also recognizes numbers as refering to the correspondingly numbered argument, i.e. it uses stkarg in this case. Thus

```
:@ FIE
FIE
:?= 2
```

will print the name and value of the second argument of FIE.

?= can also be used on brkcoms, in which case the next command on brkcoms is treated as the rest of the teletype line. For example, if brkcoms is (EVAL ?= (X Y) GO), brkexp will be evaluated, the values of X and Y printed, and then the function exited with its value being printed.

BT

Prints a backtrace of *function names only* starting at lastpos. (See discussion of @ above) The several nested calls in system packages such as break, edit, and the top level executive appear as the single entries **BREAK**, **EDITOR**, and **TOP** respectively.

BTV — Prints a backtrace of function names *with* variables beginning at lastpos.

BTV* — Same as BTV except also prints arguments of internal calls to eval. (See Section 12)

BTV! — Same as BTV except prints *everything* on stack. (See Section 12).

BT, BTV, BTV*, and BTV! all permit an optional functional argument which is a predicate that chooses functions to be *skipped* on the backtrace, e.g., BT SUBRP will skip all SUBRs, BTV (LAMBDA (X) (NOT (MEMB X FOOFNS))) will skip all but those functions on FOOFNS. If used as a brkcom the functional argument is no longer optional, i.e. the next brkcom must either be the functional argument, or NIL if no functional argument is to be applied.

For BT, BTV, BTV*, and BTV!, if control-P is used to change a printlevel during the backtrace, the printlevel will be restored after the backtrace is completed.

ARGS — Prints the names of the variables bound at lastpos, i.e. variables[lastpos] (Section 12). For most cases, these are the arguments to the function entered at that position, i.e. arglist[stkname[lastpos]].

The following two commands are for use only with unbound atoms or undefined function breaks (see Section 16).

= form, = fn[args] — *only* for the break following an unbound atom error. Sets the atom to the value of the form, or function and arguments, exits from the break returning that value, and continues the computation, e.g.

```
U.B.A.
(FOO BROKEN)
:= (COPY FIE)
```

sets FOO and goes on.

-> expr

for use either with unbound atom error, or undefined function error. Replaces the expression containing the error with expr[4] (not the value of expr) e.g.,

```
U.D.F.
(FOO1 BROKEN)
:-> FOO
```

changes the FOO1 to FOO and continues the computation.

expr need not be atomic, e.g.

```
U.B.A.
(FOO BROKEN)
:-> (QUOTE FOO)
```

For U.D.F. breaks, the user can specify a function *and* initial arguments, e.g.

```
U.D.F.
(MEMBERX BROKEN)
:-> MEMBER X
```

Note that in the case of a U.D.F. error occurring immediately following a call to apply, e.g. (APPLY X Y) where the value of x is FOO and FOO is undefined, or a U.B.A. error immediately following a call to eval, e.g. (EVAL X), where the value of x is FOO and FOO is unbound, there is no expression containing the offending atom. In this case, -> cannot operate, so ? is printed and no action taken.

EDIT

designed for use in conjunction with breaks caused by errors. Facilitates editing the expression causing the break:

```
NON-NUMERIC ARG
NIL
(IPLUS BROKEN)
:EDIT
IN FOO...
(IPLUS X Z)
EDIT
*(3 Y)
*OK
FOO
:
```

and user can continue by typing OK, EVAL, etc.

---

4 -> does not change just brkexp; it changes the function or expression containing the erroneous form. In other words, the user does not have to perform any additional editing.

This command is very simple conceptually, but complicated in its implementation by all of the exceptional cases involving inteactions with compiled functions, breaks on user functions, error breaks, breaks within breaks, et al. Therefore, we shall give the following simplified explanation which will account for 90% of the situations arising in actual usage. For those others, EDIT will print an appropriate failure message and return to the break.

EDIT begins by searching up the stack beginning at lastpos (set by @ command, initially position of the break) looking for a form, i.e. an internal call to eval. Then EDIT continues from that point looking for a call to an interpreted function, or to eval. It then calls the editor on either the EXPR or the argument to eval in such a way as to look for an expression eq to the form that it first found. It then prints the form, and permits interactive editing to begin. Note that the user can then type successive 0's to the editor to see the chain of superforms for this computation.

If the user exits from the edit with an OK, the break expression is reset, if possible, so that the user can continue with the computation by simply typing OK.[5] However, in some situations, the break expression cannot be reset. For example, if a compiled function FOO incorrectly called putd and caused the error ARG NOT ATOM followed by a break on putd, EDIT might be able to find the form headed by FOO, and also find *that* form in some higher interpreted function. But after the user corrected the problem in the FOO-form, if any, he would still not have in any way informed EDIT what to do about the immediate problem, i.e. the incorrect call to putd. However, if FOO were *interpreted* EDIT would find the putd form itself, so that when the user corrected that form, EDIT could use the new corrected form to reset the break expression. The two cases are shown below:

---

[5] Evaluating the new brkexp will involve reevaluating the form that causes the break, e.g. if (PUTD (QUOTE (FOO)) big-computation) were handled by EDIT, big-computation would be reevaluated.

```
ARG NOT ATOM                        ARG NOT ATOM
(FUM)                               (PUTD BROKEN)
(PUTD BROKEN)                       :EDIT
:EDIT                               IN FOO...
IN FIE...                           (PUTD X)
(FOO X)                             EDIT
EDIT                                *(2 (CAR X))
*(2 (CAR X))                        *OK
*OK                                 FOO
NOTE: BRKEXP NOT CHANGED            :OK
FIE                                 PUTD
:?=
U = (FUM)
:(SETQ U (CAR U))
FUM
:OK
PUTD
```

IN? similar to EDIT, but just prints parent form, and superform, but does not call editor, e.g.

```
ATTEMPT TO RPLAC NIL
T
(RPLACD BROKEN)
:IN?
FOO: (RPLACD X Z)
```

Although EDIT and IN? were designed for error breaks, they can also be useful for user breaks. For example, if upon reaching a break on his function FOO, the user determines that there is a problem in the *call* to FOO, he can edit the calling form and reset the break expression with one operation by using EDIT. The following two protocol's with and without the use of EDIT, illustrate this:

```
(FOO BROKEN)                                    (FOO BROKEN)
:?=                                             :?=
X = (A B C)                                     X = (A B C)
Y = D                                           Y = D
:BT                                             :EDIT
                                                IN FIE...
FOO                                             (FOO V U)
SETQ                                            EDIT
COND              find which function           *(SW 2 3)
PROG              FOO is called from            *OK
FIE                                             FIE        6
                  (aborted with ↑E)             :OK
:EDITF(FIE)                                     FOO
EDIT
*F FOO P
(FOO V U)         edit it
*(SW 2 3)
*OK
FIE
:(SETQ Y X)       reset X and Y
(A B C)
:(SETQQ X D)
D
:?=
X = D
Y = (A B C)       check them
:OK
FOO
```

Brkcoms

The fourth argument to break1 is brkcoms, a list of break commands that break1 interprets and executes as though they were teletype input. One can think of brkcoms as another input file which always has priority over the teletype. Whenever brkcoms=NIL, break1 reads its next command from the teletype. Whenever brkcoms is not NIL, break1 takes as its next command car[brkcoms] and sets brkcoms to cdr[brkcoms]. For example, suppose the user wished to see the value of the variable x *after* a function was evaluated. He would set up a break with brkcoms=(EVAL (PRINT X) OK), which would have the desired effect. The function trace uses brkcoms: it sets up a break with two commands; the first one prints the arguments of the function, or whatever the user specifies,

---

[6] x and y have not been changed, but brkexp has. See previous footnote.

and the second is the command GO, which causes the function to be evaluated and its value printed.

If brkcoms is not NIL, the value of a break command is not printed. If you desire to see a value, you must print it yourself, as in the above example with the command (PRINT X).

Note: whenever an error occurs, brkcoms is set to NIL, and a full interactive break occurs.

Brkfile

The break package has a facility for redirecting ouput to a file. The variable brkfile should be set to the name of the file, and the file must be opened. All output resulting from brkcoms will be output to brkfile, e.g. output due to TRACE. Output due to user typein is not affected, and will always go to the terminal. brkfile is initially T.

Breakmacros

Whenever an atomic command is given breaki that it does *not* recognize, either via brkcoms or the teletype, it searches the list breakmacros for the command. The form of breakmacros is (... (macro command_1 command_2 ... command_n) ...). If the command is defined as a macro, breaki simply appends its definition, which is a sequence of commands, to the front of brkcoms, and goes on. If the command is not contained in breakmacros, it is treated as a function or variable as before.

Example: the command ARGS could be defined by including on breakmacros: (ARGS (PRINT (VARIABLES LASTPOS T))).

15.3 Break Functions

break1[brkexp;brkwhen;brkfn;brkcoms;brktype]

is an nlambda. brkwhen determines whether a break is to occur. If its value is NIL, brkexp is evaluated and returned as the value of break1. Otherwise a break occurs and an identifying message is printed using brkfn. Commands are then taken from brkcoms or the teletype and interpreted. The commands, GO, !GO, OK, !OK, RETURN and ↑, are the only ways to leave break1. The command EVAL causes brkexp to be evaluated, and saves the value on the variable !value. Other commands can be defined for break1 via breakmacros. brktype is NIL for user breaks, INTERRUPT for control-H breaks, and ERRORX for error breaks.

For error breaks, the input buffer is cleared and saved. (For control-H breaks, the input buffer was cleared at the time the control-H was typed, see Section 16.) In both cases, if the break returns a value, i.e., is not aborted via ↑ or control-D, the input buffer will be restored (see Section 14).

breakO[fn;when;coms]

sets up a break on the function fn by redefining fn as a call to break1 with brkexp an equivalent definition of fn, and when, fn, and coms, as brkwhen, brkfn, brkcoms. Puts property BROKEN on property list of fn with value a gensym defined with the original definition. Puts property BRKINFO on property list of fn with value (BREAKO when coms) (For use in conjunction with rebreak).

Adds fn to the front of the list brokenfns. Value is fn.

If fn is non-atomic and of the form (fn1 IN fn2), break0 first calls a function which changes the name of fn1 wherever it appears inside of fn2 to that of a new function, fn1-IN-fn2, which it initially defines as fn1. Then break0 proceeds to break on fn1-IN-fn2 exactly as described above. This procedure is useful for breaking on a function that is called from many places, but where one is only interested in the call from a specific function, e.g. (RPLACA IN FOO), (PRINT IN FIE), etc. It is similar to breakin described below, but can be performed *even when FN2 is compiled* or blockcompiled, whereas breakin only works on interpreted functions.

If fn1 is not found in fn2, break0 returns the value (fn1 NOT FOUND IN fn2).

If fn1 is found in fn2, in addition to breaking fn1-IN-fn2 and adding fn1-IN-fn2 to the list brokenfns, break0 adds fn1 to the property value for the property NAMESCHANGED on the property list of fn2 and adds the property ALIAS with the value (fn2 . fn1) to the property list of fn1-IN-fn2. This will enable unbreak to recognize what changes have been made and restore the function fn2 to its original state.

If fn is nonatomic and not of the above form, break0 is called for each member of fn using the same values for when, coms, and file specified in this call to break0. This distributivity permits the user to specify complicated break conditions on several functions without excessive retyping, e.g.,

```
breakO[(FOO1 ((PRINT PRIN1) IN (FOO2 FOO3)));
          (NEQ X T);(EVAL ?= (Y Z) OK)]
```

will break on FOO1, PRINT-IN-FOO2, PRINT-IN-FOO3, PRIN1-IN-FOO2 and PRIN1-IN-FOO3.

If fn is non-atomic, the value of break0 is a list of the individual values.

break[x] is a nospread nlambda. For each atomic argument, it performs break0[atom;T]. For each list, it performs apply[BREAK0;list]. For example, break[FOO1 (FOO2 (GREATERP N 5) (EVAL))] is equivalent to break0[FOO1,T] and break0[FOO2; (GREATERP N 5); (EVAL)]

trace[x] is a nospread nlambda. For each atomic argument, it performs break0[atom;T;(TRACE ?= NIL GO)][7] For each list argument, car is the function to be traced, and cdr the forms the user wishes to see, i.e. trace performs:

```
break0[car[list];T;list[TRACE;?=; cdr[list],GO]]
```

---

[7] The flag TRACE is checked for in break1 and causes the message 'function :' to be printed instead of (function BROKEN).

For example, TRACE(FOO1 (FOO2 Y)) will cause both FOO1 and FOO2 to be traced. All the arguments of FOO1 will be printed; only the value of Y will be printed for FOO2. In the special case that the user wants to see *only* the value, he can perform TRACE((fn)). This sets up a break with commands (TRACE ?= (NIL) GO).

Note: the user can always call breakO himself to obtain combination of options of break1 not directly available with break and trace. These two functions merely provide convenient ways of calling breakO, and will serve for most uses.

Breakin

Breakin enables the user to insert a break, i.e. a call to break1, at a specified location in an interpreted function. For example, if foo calls fie, inserting a break in foo before the call to fie is similar to breaking fie. However, breakin can be used to insert breaks before or after prog labels, particular SETQ expressions, or even the evaluation of a variable. This is because breakin operates by calling the editor and actually inserting a call to break1 at a specified point *inside* of the function.

The user specifies where the break is to be inserted by a sequence of editor commands. These commands are preceded by BEFORE, AFTER, or AROUND, which breakin uses to determine what to do once the editor has found the specified point, i.e. put the call to break1 BEFORE that point, AFTER that point, or AROUND that point. For example, (BEFORE COND) will insert a break before the first occurrence of cond, (AFTER COND 2 1) will insert a break after the predicate in the first cond clause, (AFTER BF (SETQ X &)) after the *last* place X is set. Note that (BEFORE TTY:) or (AFTER TTY:) permit the user to type in commands to the editor, locate the correct point, and verify it for himself

breakin[fn;where;when;coms] breakin is an nlambda. when and coms are similar to when and coms for break0, except that if when is NIL, T is used. where specifies where in the definition of fn the call to break1 is to be inserted. (See earlier discussion).

If fn is a compiled function, breakin returns (fn UNBREAKABLE) as its value.

If fn is interpreted, breakin types SEARCHING... while it calls the editor. If the location specified by where is not found, breakin types (NOT FOUND) and exits. If it is found, breakin adds the property BROKEN-IN with value T, and the property BRKINFO with value (where when coms) to the property list of fn, and adds fn to the front of the list brokenfns.

Multiple break points, can be inserted with a single call to breakin by using a list of the form ((BEFORE ...) .. (AROUND ...)) for where. It is also possible to call break or trace on a function which has been modified by breakin, and conversely to breakin a function which has been redefined by a call to break or trace.

unbreak[x] unbreak is a nospread nlambda. It takes an indefinite number of functions modified by break, trace, or breakin and restores them to their original state by calling unbreak0. Value is list of values of unbreak0.

unbreak[] will unbreak all functions on brokenfns, in reverse order. It first sets brkinfolst to NIL.

unbreak[T] unbreaks just the first function on brokenfns, i.e., the most recently broken function.

unbreak0[fn] restores fn to its original state. If fn was not broken, value is (NOT BROKEN) and no changes are made. If fn was modified by breakin, unbreakin is called to edit it back to its original state. If fn was created from (fn1 IN fn2), i.e. if it has a property ALIAS, the function in which fn appears is restored to its original state. All dummy functions that were created by the break are eliminated. Adds property value of BRKINFO to (front of) brkinfolst.

Note: unbreak0[(fn1 IN fn2)] is allowed: unbreak0 will operate on fn1-IN-fn2 instead.

unbreakin[fn] performs the appropriate editing operations to eliminate all changes made by breakin. fn may be either the name or definition of a function. Value is fn. Unbreakin is automatically called by unbreak if fn has property BROKEN-IN with value T on its property list.

rebreak[x] is an nlambda, nospread function for rebreaking functions that were previously broken without

having to respecify the break information. For each function on _x_, _rebreak_ searches _brkinfolst_ for break(s) and performs the corresponding operation. Value is a list of values corresponding to calls to _break0_ or _breakin_. If no information is found for a particular function, value is (fn - NO BREAK INFORMATION SAVED).

rebreak[] rebreaks everything on _brkinfolst_, i.e., rebreak[] is the inverse of unbreak[].

rebreak[T] rebreaks just the first break on _brkinfolst_, i.e., the function most recently unbroken.

changename[fn;from;to] — changes all occurrences of _from_ to _to_ in _fn_. _fn_ may be compiled or blockcompiled. Value is _fn_ if _from_ was found, otherwise NIL. Does not perform any modifications of property lists. Note that _from_ and _to_ do not have to be functions, e.g. they can be names of variables, or any other literals.

virginfn[fn;flg] — is the function that knows how to restore functions to their original state regardless of any amount of breaks, breakins, advising, compiling and saving exprs, etc. It is used by _prettyprint_, _define_, and the compiler. If _flg_=NIL, as for _prettyprint_, it does not modify the definition of _fn_ in the process of producing a "clean" version of the definition, i.e. it works on a copy. If _flg_=T as for the _compiler_ and

define, it physically restores the function to its original state, and prints the changes it is making, e.g. FOO UNBROKEN, FOO UNADVISED, FOO NAMES RESTORED, etc. Value is the virgin function definition.

baktrace[pos1;pos2;skipfn;varsflg;*form*flg;allflg] prints backtrace from pos1 to pos2. If skipfn is not NIL, and skipfn[stkname[pos]] is T, pos is skipped (including all variables).

varsflg=T for backtrace a la BTV

varsflg=T, *form*flg=T - BTV*

varsflg=T, allflg=T - BTV!

Index for Section 15

Page Numbers

SECTION 16

ERROR HANDLING

16.1 Unbound Atoms and Undefined Functions

Whenever the interpreter encounters an atomic form with no binding on the push-down list, and whose value contains the atom NOBIND,[1] the interpreter calls the function faulteval. Similarly, faulteval is called when a list is encountered, car of which is not the name of a function or a function object.[2] The value returned by faulteval is used by the interpreter exactly as though it were the value of the form.

faulteval is defined to print either U.B.A., for unbound atom, or U.D.F., for undefined function, and then to call break1 giving it the offending form as brkexp.[3] Once inside the break, the user can set the atom, define the function, return a specified value for the form using the RETURN command, etc., or abort

---

1 All atoms are initialized (when they are created by the read program) with their value cells (car of the atom) NOBIND, their function cells NIL, and their property lists (cdr of the atom) NIL.

2 See Appendix 2 for complete description of INTERLISP interpreter.

3 If DWIM is enabled (and a break is going to occur), faulteval also prints the offending form (in the case of a U.B.A., the parent form) and the name of the function which contains the form. For example, if FOO contains (CONS X FIE) and FIE is unbound, faulteval prints:
U.B.A. FIE [in FOO] in (CONS X FIE). Note that if DWIM is not enabled, the user can obtain this information after he is inside the break via the IN? command.

the break using the ↑ command. If the break is exited with a value, the computation will proceed exactly as though no error had occurred.[4]

The decision over whether or not to induce a break depends on the depth of computation, and the amount of time invested in the computation. The actual algorithm is described in detail below in the section on breakcheck. Suffice it to say that the parameters affecting this decision have been adjusted empirically so that trivial type-in errors do not cause breaks, but deep errors do.

16.2 Teletype Initiated Breaks

Control-H

Section 15 on the break package described how the user could cause a break when a specified function was entered. The user can also indicate his desire to go into a break at any time while a program is running by typing control-H.[5] At the next point a function is about to be entered, the function interrupt is called instead. interrupt types INTERRUPTED BEFORE followed by the function

---

[4] A similar procedure is followed whenever apply or apply* are called with an undefined function, i.e. one whose fntyp is NIL. In this case, faultapply is called giving it the function as its first argument and the list of arguments to the function as its second argument. The value returned by faultapply is used as the value of apply or apply*. faultapply is defined to print U.D.F. and then call break1 giving it (APPLY (QUOTE fn) QUOTE args)) as brkexp. Once inside the break, the user can define the function, return a specified value, etc. If the break is exited with a value, the computation will proceed exactly as though no error had occurred. faultapply is also called for undefined function calls from compiled code.

[5] As soon as control-H is typed, INTERLISP clears and saves the input buffer, and then rings the bell, indicating that it is now safe to type ahead to the upcoming break. If the break returns a value, i.e., is not aborted via ↑ or control-D, the contents of the input buffer before the control-H was typed will be restored.

name, constructs an appropriate break expression, and then calls breaki. The user can then examine the state of the computation, and continue by typing OK, GO or EVAL, and/or retfrom back to some previous point, exactly as with a user break. Control-H breaks are thus always 'safe'. Note that control-H breaks are not affected by the depth or time of the computation. However, they *only* occur when a function is called, since it is only at this time that the system is in a "clean" enough state to allow the user to interact. Thus, if a compiled program is looping without calling any functions, or is in a I/O wait, control-H will not affect it. Control-B, however, will.

Control-B

Control-B is a stronger interruption than control-H. It effectively generates an immediate error. This error is treated like any other error except that it always causes a break, regardless of the depth or time of the computation.[6] Thus if the function FOO is looping internally, typing control-B will cause the computation to be stopped, the stack unwound to the point at which FOO was called, and then cause a break. Note that the internal variables of FOO are not available in this break, and similarly, FOO may have already produced some changes in the environment before the control-B was typed. Therefore whenever possible, it is better to use control-H instead of control-B.

Control-E

If the user wishes to *abort* a computation, without causing a break, he should type control-E. Control-E does not go through the normal error machinery of scanning the stack, calling breakcheck, printing a message, etc. as described below, but simply types a carriage-return and unwinds.

---

[6] However, setting helpflag to NIL will suppress the break. See discussion of breakcheck below.

16.3 Other Types of Errors

In addition to U.B.A. and U.D.F. errors, there are currently 28 other error types in INTERLISP, e.g. P-STACK OVERFLOW, NON-NUMERIC ARG, FILE NOT OPEN, etc. A complete list is given later in this section. When an error occurs, the decision about whether or not to break is handled by breakcheck and is the same as with U.B.A. and U.D.F. errors. If a break is to occur, the exact action that follows depends on the type of error. For example, if a break is to occur following evaluation of (RPLACA NIL (ADD1 5)) (which causes an ATTEMPT TO RPLAC NIL error), the message printed will be (RPLACA BROKEN), brkexp will be (RPLACA U V W), U will be bound to NIL, V to 6, and W to NIL, and the stack will look like the user had broken on rplaca himself. Following a NON-NUMERIC ARG error, the system will type IN followed by the name of the most recently entered function, and then (BROKEN). The system will then effectively be in a break *inside* of this function. brkexp will be a call to ERROR so that if the user types OK or EVAL or GO, a ? will be printed and the break maintained. However, if the break is exited with a value via the RETURN command,[7] the computation will proceed exactly as though no error had occurred.

16.4 Breakcheck - When to Break

The decision as to whether or not to induce a break when an error occurs is handled by the function breakcheck.[8] The user can suppress all error breaks by setting the variable helpflag to NIL (initially set to T). If helpflag=T, the decision is affected by two factors: the length of time spent in the

---

[7] Presumably the value will be a number, or the error will occur again.

[8] Breakcheck is not actually available to the user for advising or breaking since the error package is block-compiled.

computation, and the depth of the computation at the time of the error.[9] If the time is greater than helptime or the depth is greater than helpdepth, breakcheck returns T, meaning a break will occur.

Since a function is not actually entered until its arguments are evaluated,[10] the depth of a computation is defined to be the sum of the number of function calls plus the number of internal calls to eval. Thus if the user types in the expression [MAPC FOO (FUNCTION (LAMBDA (X) (COND ((NOT (MEMB X FIE)) (PRINT X] for evaluation, and FIE is not bound, at the point of the U.B.A. FIE error, two functions, mapc and cond, have been entered, and there are three internal calls to eval corresponding to the evaluation of the forms (COND ((NOT (MEMB X FIE)) (PRINT X))), (NOT (MEMB X FIE)), and (MEMB X FIE).[11] The depth is thus 5.

breakcheck begins by searching back up the parameter stack looking for an errorset.[12] At the same time, it counts the number of internal calls to eval, as indicated by pseudo-variable bindings called eval-blips. See Section 12. As soon as (if) the number of eval-blips exceeds helpdepth, breakcheck can stop searching for errorset and return T, since the position of the errorset is only needed when a break is *not* going to occur. Otherwise, breakcheck continues

---

9 Except that control-B errors always break.

10 Unless of course the function does not have its arguments evaluated, i.e. is an FEXPR, FEXPR*, CFEXPR, CFEXPR*, FSUBR or FSUBR*.

11 For complete discussion of the stack and the interpreter, see Section 12.

12 errorsets are simply markers on the stack indicating how far back unwinding is to take place when an error occurs, i.e. they segment the stack into sections such as that if an error occurs in any section, control returns to the point at which the last errorset was entered, from which NIL is returned as the value of the errorset. See page 16.14.

searching until either an errorset is found[13] or the top of the stack is reached. Breakcheck then completes the depth check by counting the number of function calls between the error and the last errorset, or the top of the stack. If the number of calls plus the number of eval-blips (already counted) is greater than or equal to helpdepth, initially set to 9,[14] breakcheck returns T. Otherwise, it records the position of the last errorset, and the value of errorset's second argument, which is used in deciding whether to print the error message, and returns NIL.

breakcheck next measures the length of time spent in the computation by subtracting the value of the variable helpclock from the value of (CLOCK 2).[15] If the difference is greater than helptime milliseconds, initially set to 1000, then a break will occur, i.e., breakcheck returns T, otherwise NIL. The variable helpclock is rebound to the current value of (CLOCK 2) for each computation typed in to lispx or to a break.

The time criterion for breaking can be suppressed by *setting* helptime to NIL (or a very big number), or by *binding* helpclock to NIL. Note that *setting* helpclock to NIL will not have any effect because helpclock is rebound by lispx and by break.

If breakcheck is NIL, i.e., a break is *not* going to occur, then if an errorset was found, NIL is returned (via retfrom) as the value of the errorset, after first printing the error message if the errorset's second argument was TRUE.

---

[13] If the second argument to the errorset is INTERNAL, the errorset is ignored and searching continues. See discussion of errorset, page 16.14.

[14] Arrived at empirically, takes into account the overhead due to lispx or break.

[15] Whose value is number of milliseconds of compute time. See Section 21.

If there was no errorset, the message is printed, and control returns to evalgt. This procedure is followed for all types of errors.

Note that for all error breaks for which a break occurs, break1 will clear and save the input buffer. If the break returns a value, i.e., is not aborted via ↑ or control-D, the input buffer will be restored as described in Section 15.

16.5 Error Types

There are currently twenty-eight error types in the INTERLISP system. They are listed below by error number. The error is set internally by the code that detects the error before it calls the error handling functions. It is also the value returned by errorn if called subsequent to that type of error, and is used by errormess for printing the error message.

Most error types will print the offending expression following the message, e.g., NON-NUMERIC ARG NIL is very common. Error type 18 (control-B) always causes a break (unless helpflag is NIL). All other errors cause breaks if breakcheck returns T.

0 NONXMEM — reference to non-existent memory. Usually indicates system is very sick.

1 — Currently not used.

2 P-STACK OVERFLOW — occurs when computation is too deep, either with respect to number of function calls, or number of variable bindings. Usually because of a non-terminating recursive computation, i.e. a bug.

3	ILLEGAL RETURN	call to return when not inside of an interpreted prog.
4	ILLEGAL ARG - PUTD	second argument to putd (the definition) is not NIL, a list, or a pointer to compiled code.
5	ARG NOT ATOM - SET	first argument to set, setq, or setqq (name of the variable) is not a literal atom.
6	ATTEMPT TO SET NIL	via set or setq
7	ATTEMPT TO RPLAC NIL	attempt either to rplaca or to rplacd NIL with something other than NIL.
8	UNDEFINED OR ILLEGAL GO	go when not inside of a prog, or go to nonexistent label.
9	FILE WON'T OPEN	From infile or outfile, Section 14.
10	NON-NUMERIC ARG	a numeric function e.g. iplus, itimes, igreaterp, expected a number.
11	ATOM TOO LONG	≥ 100 characters.
12	ATOM HASH TABLE FULL	no room for any more (new) atoms.
13	FILE NOT OPEN	from an I/O function, e.g. read, print, closef.
14	ARG NOT ATOM	
15	TOO MANY FILES OPEN	≥ 16 including teletype.

16 END OF FILE — from an input function, e.g. *read*, *readc*, *ratom*. Note: the file will then be closed.

17 ERROR — call to *error*.

18 BREAK — control-B was typed.

19 ILLEGAL STACK ARG — a stack function expected a stack position and was given something else. This might occur if the arguments to a stack function are reversed. Also occurs if user specified a stack position with a function name, and that function was not found on the stack. See Section 12.

20 FAULT IN EVAL — artifact of bootstrap. Never occurs after *faulteval* has been defined as described earlier.

21 ARRAYS FULL — system will first initiate a GC: 1, and if no array space is reclaimed, will then generate this error.

22 DIRECTORY FULL — no new files can be created until user deletes some old ones and expunges.

23 FILE NOT FOUND — file name does not correspond to a file in the corresponding directory. Can also occur if file name is ambiguous.

24 FILE INCOMPATIBLE - SYSIN — from *sysin*, Section 14.

25 UNUSUAL CDR ARG LIST — a form ends in a non-list other than NIL, e.g. (CONS T . 3).

26 HASH TABLE FULL	see hash link functions, Section 7.
27 ILLEGAL ARG	Catch-all error. Currently used by evala, arg, funarg, allocate, rplstring, and sfptr.
28 ARG NOT ARRAY	elt or seta given an argument that is not a pointer to the beginning of an array.
29 ILLEGAL OR IMPOSSIBLE BLOCK	From getblk or relblk. See Section 21.
30	currently not used.
31 LISTS FULL	following a GC: 8, if a sufficient amount of list words have not been collected, and there is no un-allocated space left in the system, this error is generated.

Many system functions, e.g. define, arglist, advise, log, expt, etc, also generate errors with appropriate messages by calling error (see page 16.12) which causes an error of type 16.

Error handling by error type

Occasionally the user may want to treat certain error types different than others, e.g. always break, never break, or perhaps take some corrective action. This can be accomplished via errortypelst. errortypelst is a list of elements of the form (n expression), where n is one of the 28 error numbers. After breakcheck has been completed, but before any other action is taken, errortypelst is searched for an element with the same error number as that

causing the error. If one is found, and the evaluation of the corresponding expression produces a non-NIL value, the value is substituted for the offender, and the function causing the error is reeentered.

For this application, the following three variables may be useful:

errormess — car is the error number, cadr the "offender" e.g. (10 NIL) corresponds to NON-NUMERIC ARG NIL error.

errorpos — position of the function in which the error occurred, e.g. stkname[errorpos] might be IPLUS, RPLACA, INFILE, etc.

breakchk — value of breakcheck, i.e. T means a break will occur, NIL means one will not.

For example, putting

```
[10 (AND (NULL (CADR ERRORMESS))
         (SELECTQ (STKNAME ERRORPOS)
                  ((IPLUS ADD1 SUB1) 0)
                  (ITIMES 1)
                  (PROGN (SETQ BREAKCHK T) NIL]
```

on errortypelst would specify that whenever a NON-NUMERIC ARG - NIL error occurred, and the function in question was IPLUS, ADD1, or SUB1, 0 should be used for the NIL. If the function was ITIMES, 1 should be used. Otherwise, always break. Note that the latter case is achieved not by the value returned, but by the *effect* of the evaluation, i.e. setting BREAKCHK to T. Similarly, (16 (SETQ BREAKCHK NIL)) would prevent END OF FILE errors from ever breaking.

16.6 Error Functions

errorx[erxm]

is the entry to the error routines. If erxm=NIL, errorn[] is used to determine the error-message. Otherwise, seterrorn[erxm] is performed, 'setting' the error type and argument. Thus following either errorx[(10 T)] or (PLUS T), errorn[] is (10 T). errorx calls breakcheck, and either induces a break or prints the message and unwinds to the last errorset. Note that errorx can be called by any program to intentionally induce an error of any type. However, for most applications, the function error will be more useful.

error[mess1;mess2;nobreak]

The message that is (will be) printed is mess1 (using prin1), followed by a space if mess1 is an atom, otherwise a carriage return. Then mess2 is printed, using prin1 if mess2 is a string, otherwise print. e.g., error["NON-NUMERIC ARG";T] will print

```
NON-NUMERIC ARG
T
```

and error[FOO;"NOT A FUNCTION"] will print FOO NOT A FUNCTION. (If both mess1 and mess2 are NIL, the message is simply ERROR.)

If nobreak=T, error prints its message and then calls error!. Otherwise it calls errorx[(17 (mess1 . mess2))], i.e. generates an error of type 17, in which case the decision as to

whether or not to break, and whether or not to print a message, is handled as per any other error.

help[mess1;mess2] — prints mess1 and mess2 a la error, and then calls break1. If both mess1 and mess2 are NIL, HELP! is used for the message. help is a convenient way to program a default condition, or to terminate some protion of a program which theoretically the computation is never supposed to reach.

error![][16] — programmable control-E, i.e., immediately returns from last errorset or resets.

reset[] — Programmable control-D, i.e. immediately returns to the top level.

errorn[] — returns information about the last error in the form (n x) where n is the error type number and x is the expression which was (would have been) printed out after the error message. Thus following (PLUS T), errorn[] is (10 T).

errormess[u] — prints message corresponding to an errorn that yielded u. For example, errormess[(10 T)] would print

```
NON-NUMERIC ARG
T
```

---

16 Pronounced "error-bang".

errorset[u;v][17]

performs eval[u]. Note that errorset is a lambda-type of function, and that its arguments are evaluated *before* it is entered, i.e. errorset[x] means eval is called with the *value* of x. In most cases, ersetq and nlsetq (described below) are more useful. If no error occurs in the evaluation of u, the value of errorset is a list containing one element, the value of eval[u]. If an error did occur, the value of errorset is NIL.

The argument v controls the printing of error messages if an error occurs. If v=T, the error message is printed; if v=NIL it is not.

If v=INTERNAL, the errorset is ignored for the purpose of deciding whether or not to break or print a message. However, the errorset is in effect for the purpose of flow of control, i.e. if an error occurs, this errorset returns NIL.

ersetq[ersetx]

nlambda, performs errorset[ersetx;t], i.e. (ERSETQ (FOO)) is equivalent to (ERRORSET (QUOTE (FOO)) T).

nlsetq[nlsetx]

nlambda, performs errorset[nlsetx;NIL].

---

[17] errorset is a subr, so the names "u" and "v" don't actually appear on the stack nor will they affect the evaluation.

Index for Section 16

Page Numbers

SECTION 17

AUTOMATIC ERROR CORRECTION - THE DWIM FACILITY[1]

17.1 Introduction

A surprisingly large percentage of the errors made by INTERLISP users are of the type that could be corrected by another LISP programmer without any information about the purpose of the program or expression in question, e.g. misspellings, certain kinds of parentheses errors, etc. To correct these types of errors we have implemented in INTERLISP a DWIM facility, short for DO-What-I-Mean. DWIM is called automatically whenever an error[2] occurs in the evaluation of an INTERLISP expression. DWIM then proceeds to try to correct the mistake using the current context of computation plus information about what the user had previously been doing, (and what mistakes he had been making) as guides to the remedy of the error. If DWIM is able to make the correction, the computation continues as though no error had occurred. Otherwise, the procedure is the same as though DWIM had not intervened: a break occurs, or an unwind to the last errorset, as described in Section 16. The following protocol illustrates the operation of DWIM.

---

1 DWIM was designed and implemented by W. Teitelman. It is discussed in [Tei2].

2 Currently, DWIM only operates on unbound atoms and undefined function errors.

Example

The user defines a function fact of one argument, n. The value of fact[n] is to be n factorial.

```
←DEFINEQ((FACT (LAMBDA (N) (COND
((ZEROP N9 1) ((T (ITIMES N (FACT 8SUB1 N]
(FACT)
←
```

Note that the definition of fact contains several mistakes: Itimes and fact have been misspelled; the 9 in N9 was intended to be a right parenthesis, but the shift key was not depressed; similarly, the 8 in 8SUB1 was intended to be a left parenthesis; and finally, there is an extra left parenthesis in front of the T that begins the final clause in the conditional.

```
←PRETTYPRINT((FACCT]                                    [1]
=PRETTYPRINT                                            [2]
=FACT                                                   [3]

(FACT
  [LAMBDA (N)
    (COND
      ((ZEROP N9 1)
        ((T (ITIMS N (FACCT 8SUB1 N])
(FACT)
←
```

After defining fact, the user wishes to look at its definition using PRETTYPRINT, which he unfortunately misspells.[1] Since there is no function PRETTYPRINT in the system, a U.D.F. error occurs, and DWIM is called. DWIM invokes its spelling corrector, which searches a list of functions frequently used (by *this* user) for the best possible match. Finding one that is extremely close, DWIM proceeds on the assumption that PRETTYPRINT meant PRETTYPRINT, notifies the user of this, [2] and calls prettyprint.

At this point, PRETTYPRINT would normally print (FACCT NOT PRINTABLE) and exit, since facct has no definition. Note that this is *not* an INTERLISP error

condition, so that DWIM would not be called as described above. However, it is obviously not what the user *meant*.

This sort of mistake is corrected by having prettyprint itself explicitly invoke the spelling corrector portion of DWIM whenever given a function with no expr definition. Thus with the aid of DWIM, prettyprint is able to determine that the user wants to see the definition of the function fact,[3] and proceeds accordingly.

```
←FACT(3]                                            [4]
N9 [IN FACT] -> N ) ?  YES
[IN FACT] (COND -- ((T --))) ->
          (COND -- (T --))
ITIMS [IN FACT] -> ITIMES                           [5]
FACCT [IN FACT] -> FACT
8SUB1 [IN FACT] ->  ( SUB1 ?  YES
6
←PP FACT                                            [6]

(FACT
  [LAMBDA (N)
    (COND
      ((ZEROP N)
        1)
      (T (ITMES N (FACT (SUB1 N])
FACT
←
```

The user now calls his function fact.[4] During its execution, five errors occur, and DWIM is called five times.[5] At each point, the error is corrected, a message printed describing the action taken, and the computation allowed to continue as if no error had occurred. Following the last correction, 6 is printed, the value of fact(3). Finally, the user prettyprints the new, now correct, definition of fact.[6]

In this particular example, the user was shown operating in TRUSTING mode, which gives DWIM carte blanche for most corrections. The user can also operate in CAUTIOUS mode, in which case DWIM will inform him of intended corrections before they are made, and allow the user to approve or disapprove of them. For

most corrections, if the user does not respond in a specified interval of time, DWIM automatically proceeds with the correction, so that the user need intervene only when he does not approve. Sample output is given below. Note that the user responded to the first, second, and fifth questions; DWIM responded for him on the third and fourth.

```
←FACT(3)
N9 [IN FACT] -> N ) ?  YES                          [1]
U.D.F. T [IN FACT]   FIX?   YES                     [2]
[IN FACT] (COND -- ((T --))) ->
          (COND -- (T --))
ITIMS [IN FACT] -> ITIMES ?   ...YES                [3]
FACCT [IN FACT] -> FACT ?   ...YES                  [4]
8SUB1 [IN FACT] ->  ( SUB1 ?  NO                    [5]
U.B.A.
(8SUB1 BROKEN)
:
```

We have put a great deal of effort into making DWIM 'smart', and experience with perhaps fifty different users indicates we have been very successful; DWIM seldom fails to correct an error the user feels it should have, and almost never mistakenly corrects an error. However, it is important to note that even when DWIM *is* wrong, no harm is done:[3] since an error had occurred, the user would have had to intervene anyway if DWIM took no action. Thus, if DWIM mistakenly corrects an error, the user simply interrupts or aborts the computation, UNDOes the DWIM change using UNDO described in Section 22, and makes the correction he would have had to make without DWIM. It is this benign quality of DWIM that makes it a valuable part of INTERLISP.

---

[3] Except perhaps if DWIM's correction mistakenly caused a destructive computation to be initiated, and information was lost before the user could interrupt. We have not yet had such an incident occur.

17.2 Interaction with DWIM

DWIM is enabled by performing either DWIM[C], for CAUTIOUS mode, or DWIM[T] for TRUSTING mode.[4] In addition to setting dwimflg to T and redefining faulteval and faultapply as described on page 17.14, DWIM[C] sets approveflg to T, while DWIM[T] sets approveflg to NIL. The setting of approveflg determines whether or not the user wishes to be asked for approval before a correction that will modify the definition of one of his functions. In CAUTIOUS mode, i.e. approveflg=T, DWIM will ask for approval; in TRUSTING mode, DWIM will not. For corrections to expressions typed in by the user for immediate execution,[5] DWIM always acts as though approveflg were NIL, i.e. no approval necessary.[6] In either case, DWIM always informs the user of its action as described below.

Spelling Correction Protocol

The protocol used by DWIM for spelling corrections is as follows: If the correction occurs in type-in, print = followed by the correct spelling, followed by a carriage-return, and then continue, e.g.

---

[4] INTERLISP arrives with DWIM enabled in CAUTIOUS mode. DWIM can be disabled by executing DWIM[] or by setting dwimflg to NIL. See page 17.23.

[5] Typed into lispx. lispx is used by evalqt and break, as well as for processing the editor's E command. Functions that call the spelling corrector directly, such as editdefault (Section 9), specify whether or not the correction is to be handled as type-in. For example, in the case of editdefault, commands typed directly to the editor are treated as type-in, so that corrections to them will never require approval. Commands given as an argument to the editor, or resulting from macro expansions, or from IF, LP, ORR commands etc. are not treated as type-in, and thus approval will be requested if approveflg=T.

[6] For certain types of corrections, e.g. run-on spelling corrections, 8-9 errors, etc., dwim always asks for approval, regardless of the setting of approveflg.

```
user types:     ←(SETQ FOO (NCOCN FIE FUM))
DWIM types:     =NCONC
```

If the correction does not occur in type-in, print the incorrect spelling, followed by [IN function-name], ->, and then the correct spelling, e.g. ITIMS [IN FACT] -> ITIMES as shown on page 17.3.[7] Then, if approveflg=NIL, print a carriage return, make the correction and continue. Otherwise, print a few spaces and a ? and then wait for approval.[8] The user then has six options. He can:

1. Type Y; DWIM types ES, and proceeds with the correction.
2. Type N; DWIM types O, and does not make the correction.
3. Type ↑; DWIM does not make the correction, and furthermore guarantees that the error will not cause a break. See footnote on page 17.15.
4. Type control-E; for error correction, this has the same effect as typing N.
5. Do nothing; in which case DWIM will wait a specified interval,[9] and if the user has not responded, DWIM will type ... followed by the default answer.[10]
6. Type space or carriage-return; in which case DWIM will wait indefinitely. This option is intended for those cases where the user wants to think about his answer, and wants to insure that DWIM does not get 'impatient' and answer for him.

---

[7] The appearance of -> is to call attention to the fact that the user's function will be or has been changed.

[8] Whenever an interaction is about to take place and the user has typed ahead, DWIM types several bells to warn the user to stop typing, then clears and saves the input buffers, restoring them after the interaction is complete. Thus if the user has typed ahead before a DWIM interaction, DWIM will not confuse his type ahead with the answer to its question, nor will his type ahead be lost.

[9] Equal to dwimwait seconds. DWIM operates by dismissing for 500 milliseconds, then checking to see if anything has been typed. If not, it dismisses again, etc. until dwimwait seconds have elapsed. Thus, there will be a delay of at most 1/2 second before DWIM responds to the user's answer.

[10] The default is always YES unless otherwise stated.

The procedure for spelling correction on other than INTERLISP errors is analogous. If the correction is being handled as type-in, DWIM prints = followed by the correct spelling, and returns it to the function that called DWIM, e.g. =FACT as shown on page 17.2. Otherwise, DWIM prints the incorrect spelling, followed by the correct spelling. Then if approveflg=NIL, DWIM prints a carriage-return and returns the correct spelling. Otherwise, DWIM prints a few spaces and a ? and then waits for approval. The user can then respond with Y, N, control-E, space, carriage return, or do nothing as described.

Note that since the spelling corrector itself is not errorset protected, typing N and typing control-E may have different effects when the spelling corrector is called directly.[11] The former simply instructs the spelling corrector to return NIL, and lets the calling function decide what to do next; the latter causes an error which unwinds to the last errorset, however far back that may be.

Parentheses Errors Protocol

As illustrated earlier on page 17.3, DWIM will correct errors consisting of typing 8 for left parenthesis and 9 for right parenthesis. In these cases, the interaction with the user is similar to that for spelling correction. If the error occurs in type-in, DWIM types = followed by the correction, e.g.

```
user types:    ←(SETQ FOO 8CONS FIE FUM]
DWIM types:    = ( CONS
lispx types:   (A B C D)
```

Otherwise, DWIM prints the offending atom, [IN function-name], ->, the proposed

---

[11] The DWIM error correction routines are errorset protection.

correction, a few spaces and a ?, and then waits for approval, e.g. N9 [IN FACT] -> N) ? as shown on page 17.3. The user then has the same six options as for spelling correction.[12] If the user types Y, DWIM then operates exactly the same as when approveflg=NIL, i.e. makes the correction and prints its message.

U.D.F. T Errors Protocol

DWIM corrects certain types of parentheses errors involving a T clause in a conditional, namely errors of the form:

1. (COND --) (T --), i.e. the T clause appears outside and immediately following the COND;

2. (COND -- (-- & (T --))), i.e. the T clause appears inside a previous clause; and

3. (COND -- ((T --))), i.e. the T clause has an extra pair of parentheses around it.[13]

If the error occurs in type-in, DWIM simply types T FIXED and makes the correction. Otherwise if approveflg=NIL, DWIM makes the correction, and prints a message consisting of [IN function-name], followed by one of the above incorrect forms of COND, followed by ->, then on the next line the corresponding correct form of the COND, e.g.

---

12 except the waiting time is 3*dwimwait seconds.

13 For U.D.F. T errors that are not one of these three types, DWIM takes no corrective action at all, i.e. the error will occur.

```
[IN FACT] (COND -- ((T --))) ->
          (COND -- (T --))
```

as shown on page 17.3.

If approveflg=T, DWIM prints U.D.F. T, followed by [IN function-name], several spaces, and then FIX? and waits for approval. The user then has the same options as for spelling corrections and parenthesis errors. If the user types Y or defaults, DWIM then proceeds exactly the same as when approveflg=NIL, i.e. makes the correction and prints its message, as shown on page 17.4.

Having made the correction, DWIM must then decide how to proceed with the computation. In case 1, (COND --) (T --), DWIM cannot know whether the last clause of the COND before the T clause succeeded or not, i.e. if the T clause had been inside of the COND, would it have been entered? Therefore DWIM asks the user 'CONTINUE WITH T CLAUSE' (with a default of YES). If the user types N, DWIM continues with the form after the COND, i.e. the form that originally followed the T clause.

In case 2, (COND -- (-- & (T --))), DWIM has a different problem. After moving the T clause to its proper place, DWIM must return as the value of the COND, the value of the expression corresponding to &. Since this value is no longer around, DWIM asks the user, 'OK TO REEVALUATE' and then prints the expression corresponding to &. If the user types Y, or defaults, DWIM continues by reevaluating &, otherwise DWIM aborts, and a U.D.F. T error will then occur (even though the COND has in fact been fixed).[14]

---

[14] If DWIM can determine for itself that the form can safely be reevaluated, it does not consult the user before reevaluating. DWIM can do this if the form is atomic, or car of the form is a member of the list okreevalst, and each of the arguments can safely be reevaluated, e.g. (SETQ X (CONS (IPLUS Y Z) W)) is safe to reevaluate because SETQ, CONS, and IPLUS are all on okreevalst.

In case 3, (COND -- ((T --))), there is no problem with continuation, so no further interaction is necessary.

17.3 Spelling Correction

The spelling corrector is given as arguments a misspelled word (word means literal atom), a spelling list (a list of words), and a number: xword, splst, and rel respectively. Its task is to find that word on splst which is closest to xword, in the sense described below. This word is called a *respelling* of xword. rel specifies the minimum 'closeness' between xword and a respelling. If the spelling corrector cannot find a word on splst closer to xword than rel, or if it finds two or more words equally close, its value is NIL, otherwise its value is the respelling.[15]

The exact algorithm for computing the spelling metric is described later on page 17.20, but briefly 'closeness' is inversely proportional to the number of disagreements between the two words, and directly proportional to the length of the longer word, e.g. PRTTYPRNT is 'closer' to PRETTYPRINT than CS is to CONS even though both pairs of words have the same number of disagreements. The spelling corrector operates by proceeding down splst, and computing the closeness between each word and xword, and keeping a list of those that are closest. Certain differences between words are not counted as disagreements, for example a single transposition, e.g. CONS to CNOS, or a doubled letter, e.g. CONS to CONSS, etc. In the event that the spelling corrector finds a word on splst with *no* disagreements, it will stop searching and return this word as the respelling. Otherwise, the spelling corrector continues through the entire

---

[15] The spelling corrector can also be given an optional functional argument, fn, to be used for selecting out a subset of splst, i.e. only those members of splst that satisfy fn will be considered as possible respellings.

spelling list. Then if it has found one and only one 'closest' word, it returns this word as the respelling. For example, if _xword_ is VONS, the spelling corrector will probably return CONS as the respelling. However, if _xword_ is CONZ, the spelling corrector will not be able to return a respelling, since CONZ is equally close to both CONS and COND. If the spelling corrector finds an acceptable respelling, it interacts with the user as described earlier.

In the special case that the misspelled word contains one or more alt-modes, the spelling corrector operates somewhat differently. Instead of trying to find the closest word as above, the spelling corrector searches for those words on _splst_ that match _xword_, where an alt-mode can match any number of characters (including 0), e.g. FOO$ matches FOO1 and FOO, but not NEWFOO. FOO matches all three. In this case, the entire spelling list is always searched, and if more than one respelling is found, the spelling corrector prints AMBIGUOUS, and returns NIL. For example, CON$ would be ambiguous if both CONS and COND were on the spelling list. If the spelling corrector finds one and only one respelling, it interacts with the user as described earlier.

For both spelling correction and spelling completion, regardless of whether or not the user approves of the spelling corrector's choice, the respelling is moved to the front of _splst_. Since many respellings are of the type with no disagreements, this procedure has the effect of considerably reducing the time required to correct the spelling of frequently misspelled words.

Spelling Lists

Although any list of atoms can be used as a spelling list, e.g. _editcomsa_, _brokenfns_, _filelst_, etc., four lists are maintained especially for spelling

correction: spellings1, spellings2, spellings3, and userwords.[16]

Spellings1 is a list of functions used for spelling correction when an input is typed in apply format, and the function is undefined, e.g. EDTIF(FOO). Spellings1 is initialized to contain defineq, break, makefile, editf, tcompl, load, etc. Whenever lispx is given an input in apply format, i.e. a function and arguments, the name of the function is added to spellings1.[17] For example, typing CALLS(EDITF) will cause CALLS to be added to spellings1. Thus if the user typed CALLS(EDITF) and later typed CALLLS(EDITV), since spellings1 would then contain CALLS, DWIM would be successful in correcting CALLLS to CALLS.[18]

Spellings2 is a list of functions used for spelling correction for all other undefined functions. It is initialized to contain functions such as add1, append, cond, cons, go, list, nconc, print, prog, return, setq, etc. Whenever lispx is given a non-atomic form, the name of the function is added to spellings2. For example, typing (RETFROM (STKPOS (QUOTE FOO) 2)) to a break would add retfrom to spellings2. Function names are also added to spellings2 by define, defineq, load (when loading compiled code), unsavedef, editf, and prettyprint.

Spellings3 is a list of words used for spelling correction on all unbound atoms. Spellings3 is initialized to editmacros, breakmacros, brokenfns, and advisedfns. Whenever lispx is given an atom to evaluate, the name of the atom

---

16 All of the remarks on maintaining spelling lists apply *only* when DWIM is enabled, as indicated by dwimflg=T.

17 Only if the function has a definition.

18 If CALLLS(EDITV) were typed before CALLS had been 'seen' and added to spellings1, the correction would not succeed. However, the alternative to using spelling lists is to search the entire oblist, a procedure that would make spelling correction intolerably slow.

is added to spellings3.[19] Atoms are also added to spellings3 whenever they are edited by editv, and whenever they are set via rpaq or rpaqq. For example, when a file is loaded, all of the variables set in the file are added to spellings3. Atoms are also added to spellings3 when they are set by a lispx input, e.g. typing (SETQ FOO (REVERSE (SETQ FIE --))) will add both FOO and FIE to spellings3.

Userwords is a list containing both functions and variables that the user has *referred* to e.g. by breaking or editing. Userwords is used for spelling correction by arglist, unsavedef, prettyprint, break, editf, advise, etc. Userwords is initially NIL. Function names are added to it by define, defineq, load, (when loading compiled code, or loading exprs to property lists) unsavedef, editf, editv, editp, prettyprint, etc. Variable names are added to userwords at the same time as they are added to spellings3. In addition, the variable lastword is always set to the last word added to userwords, i.e. the last function or variable referred to by the user, and the respelling of NIL is defined to be the value of lastword. Thus, if the user has just defined a function, he can then edit it by simply typing EDITF(), or prettyprint it by typing PP().

Each of the above four spelling lists are divided into two sections separated by a NIL. The first section contains the 'permanent' words; the second section contains the temporary words. New words are added to the corresponding spelling list at the front of its temporary section.[20] (If the word is already in the temporary section, it is moved to the front of that section; if the word is in the permanent section, no action is taken.) If the length of the

---

[19] Only if the atom has a value other than NOBIND.

[20] Except that functions added to spellings1 or spellings2 by lispx are always added to the end of the permanent section.

temporary section then exceeds a specified number, the last (oldest) word in the temporary section is forgotten, i.e. deleted. This procedure prevents the spelling lists from becoming cluttered with unimportant words that are no longer being used, and thereby slowing down spelling correction time. Since the spelling corrector moves each word selected as a respelling to the front of its spelling list, the word is thereby moved into the permanent section. Thus once a word is mispelled and corrected, it is considered important and will never be forgotten.

The maximum length of the temporary section for spellings1, spellings2, spellings3 and userwords is given by the value of #spellings1, #spellings2, #spellings3, and #userwords, initialized to 30, 30, 30, and 60 respectively. Using these values, the average length of time to search a spelling list for one word is about 4 milliseconds.[21]

17.4 Error Correction

As described in Section 16, whenever the interpreter encounters an atomic form with no binding, or a non-atomic form car of which is not a function or function object, it calls the function faulteval. Similarly, when apply is given an undefined function, it calls faultapply. When DWIM is enabled, faulteval and faultapply are redefined to first call dwimblock, a part of the DWIM package. If the user aborts by typing control-E, or if he indicates disapproval of DWIM's intended correction by answering N as described on page

---

[21] If the word is at the front of the spelling list, the time required is only 1 millisecond. If the word is not on the spelling list, i.e. the entire list must be searched, the time is proportional to the length of the list; to search a spelling list of length 60 takes about 7 milliseconds.

17.6, or if DWIM cannot decide how to fix the error, dwimblock returns NIL.[22] In this case, faulteval and faultapply proceed exactly as described in Section 16, by printing a U.B.A. or U.D.F. message, and going into a break if the requirements of breakcheck are met, otherwise unwinding to the last errorset.

If DWIM can (and is allowed to) correct the error, dwimblock exits by performing reteval of the corrected form, as of the position of the call to faulteval or faultapply. Thus in the example at the beginning of the chapter, when DWIM determined that ITIMS was ITIMES misspelled, DWIM called reteval with (ITIMES N (FACCT 8SUB1 N)). Since the interpreter uses the value returned by faulteval exactly as though it were the value of the erroneous form, the computation will thus proceed exactly as though no error had occurred.

In addition to continuing the computation, DWIM also repairs the cause of the error whenever possible.[23] Thus in the above example, DWIM also changed (with rplaca) the expression (ITIMS N (FACCT 8SUB1 N)) that caused the error.

Error correction in DWIM is divided into three categories: unbound atoms, undefined cars of form, and undefined function in apply. Assuming that the user approves if he is asked, the action taken by DWIM for the various types of errors in each of these categories is summarized below. The protocol of DWIM's interaction with the user has been described earlier.

---

[22] If the user answers with ↑, (see page 17.6) dwimblock is exited by performing reteval[FAULTEVAL;(ERROR!)], i.e. an error is generated at the position of the call to faulteval.

[23] If the user's program had *computed* the form and called eval, e.g. performed (EVAL (LIST X Y)) and the value of x was a misspelled function; it would not be possible to repair the cause of the error, although DWIM could correct the misspelling each time it occurred.

Unbound Atoms

1. If the first character of the unbound atom is ', DWIM assumes that the user (intentionally) typed 'atom for (QUOTE atom) and makes the appropriate change. No message is typed, and no approval requested.

   If the unbound atom is just ' itself, DWIM assumes the user wants the *next* expression quoted, e.g. (CONS X '(A B C)) will be changed to (CONS X (QUOTE (A B C))). Again no message will be printed or approval asked. (If no expression follows the ', DWIM gives up.)

2. If CLISP (Section 23) is enabled, and the atom is part of a CLISP construct, the CLISP transformation is performed and the result returned, e.g. N-1 is transformed to (SUB1 N).

3. If the atom contains an 8, DWIM assumes the 8 was intended to be a left parenthesis, and calls the editor to make appropriate repairs on the expression containing the atom. DWIM assumes that the user did not notice the mistake, i.e. that the entire expression was affected by the missing left parenthesis. For example, if the user types (SETQ X (LIST (CONS 8CAR Y) (CDR Z)) Y), the expression will be changed to (SETQ X (LIST (CONS (CAR Y) (CDR Z)) Y)).

   The 8 does not have to be the first character of the atom, e.g. DWIM will handle (CONS X8CAR Y) correctly.

4. If the atom contains a 9, DWIM assumes the 9 was intended to be a right parenthesis and operates as in number 3.

5. If the atom begins with a 7, the 7 is treated as a ', e.g. 7FOO becomes 'FOO, and then (QUOTE FOO).

6. If the atom is an edit command (a member of editcomsa), and the error occurred in type-in, the effect is the same as though the user typed EDITF(), followed by the atom, i.e. DWIM assumes the user wants to be in the editor editing the last thing he referred to. Thus, if the user defines the function foo and then types P, he will see =FOO, followed by EDIT, followed by the printout associated with the execution of the P command, followed by *, at which point he can continue editing foo.

7. If dwimuserfn=T, DWIM calls dwimuserfn, and if it returns a non-NIL value, DWIM returns this value. dwimuserfn is discussed below.

8. If the unbound atoms occurs in a function, DWIM attempts spelling correction using as a spelling list the list of lambda and prog variables of the function.

9. If the unbound atom occurred in a type-in to a break, DWIM attempts spelling correction using the lambda and prog variables of the broken function.

10. Otherwise, DWIM attempts spelling correction using spellings3.

If all fail, DWIM gives up.

Undefined car of Form

1. If car of the form is T, DWIM assumes a misplaced T clause and operates as described on page 17.8.

2. If car of the form is F/L, DWIM changes the F/L to FUNCTION(LAMBDA,e.g. (F/L (Y) (PRINT (CAR Y))) is changed to (FUNCTION (LAMBDA (Y) (PRINT (CAR Y))). No message is printed and no approval requested. If the user omits the variable list, DWIM supplies (X), e.g. (F/L (PRINT (CAR X))) becomes (FUNCTION (LAMBDA (X) (PRINT (CAR X)))). DWIM determines that the user has supplied the variable list when more than one expression follows F/L, car of the first expression is not the name of a function, and every element in the first expression is atomic. For example, DWIM will supply (X) when correcting (F/L (PRINT (CDR X)) (PRINT (CAR X))).

3. If car of the form is IF, if, or one of the CLISP iterative statement operators, e.g. FOR, WHILE, DO et al, the indicated transformation is performed, and the result returned as the corrected form.

4. If car of the form has a function definition, DWIM attempts spelling correction on car of the definition using the spelling list (LAMBDA NLAMBDA).

5. If car of the form has an EXPR property, DWIM prints car of the form, followed by 'UNSAVED', performs an unsavedef, and continues. No approval is requested.

6. If car of the form has a property FILEDEF, the definition is to be found on a file. If the value of the property is atomic, the entire file is to be loaded. If a list, car is the name of the file and cdr the relevant functions, and loadfns will be used. DWIM first checks to see if the file appears in the attached directory, <NEWLISP>'s directory, or <LISP>'s directory, and if found, types "SHALL I LOAD" followed by the file name or list of functions. If the user approves, DWIM loads the function(s) or file, and continues the computation. edita, breakdown, circlmaker, cplists, and the pattern match compiler and record capability of CLISP are implemented in this fashion.

7. If CLISP is enabled, and car of the form is part of a CLISP construct, the indicated transformation is performed, e.g. (N←N-1) becomes (SETQ N (SUB1 N)).

8. If car of the form contains an 8, DWIM assumes a left parenthesis was intended e.g. (CONS8CAR X).

9. If car of the form contains a 9, DWIM assumes a right parenthesis was intended.

10. If car of the form is a list, DWIM attempts spelling correction on caar of the form using spelling list (LAMBDA NLAMBDA). If successful, DWIM returns the corrected expression itself.

11. If car of the form is a small number, and the error occurred in type-in, DWIM assumes the form is really an edit command and operates as described in case 6 of unbound atoms.

12. If car of the form is an edit command (a member of editcomsl), DWIM operates as in 11.

13. If dwimuserfn=T, dwimuserfn is called, and if it returns a non-NIL value, DWIM returns this value.

14. If the error occurs in a function, or in a type-in while in a break, DWIM checks to see if the last character in car of the form is one of the lambda or prog variables, and if the first n-1 characters are the name of a defined function, and if so makes the corresponding change, e.g. (MEMBERX Y) will be changed to (MEMBER X Y). The protocol followed will be the same as for that of spelling correction, e.g. if approveflg=T, DWIM will type MEMBERX [IN FOO] -> MEMBER X?

15. Otherwise, DWIM attempts spelling correction using spellings2 as the spelling list.

If all fail, DWIM gives up.

Undefined Function in Apply

1. If the function has a definition, DWIM attempts spelling correction on car of the definition using the spelling list (LAMBDA NLAMBDA).

2. If the function has an EXPR property, DWIM prints its name followed by 'UNSAVED', performs an unsavedef and continues. No approval is requested.

3. If the function has a property FILEDEF, DWIM proceeds as in case 6 of undefined car of form.

4. If the error resulted from type-in, and CLISP is enabled, and the function name contains a CLISP operator, DWIM performs the indicated transformation, e.g. the user types FOO←(APPEND FIE FUM).

5. If the function name contains an 8, DWIM assumes a left parenthesis was intended, e.g. EDIT8FOO].

6. If the 'function' is a list, DWIM attempts spelling correction on car of the list using the spelling list (LAMBDA NLAMBDA).

7. If the function is a number and the error occurred in type-in, DWIM assumes the function is an edit command, and operates as described in case 6 of unbound atoms, e.g. the user types (on one line) 3 -1 P.

8. If the function is the name of an edit command (on either editcomsa or editcomsl), DWIM operates as in 7, e.g. user types F COND.

9. If dwimuserfn=T, dwimuserfn is called, and if it returns a non-NIL value, this value is treated as the *form* used to continue the computation.

10. Otherwise DWIM attempts spelling correction using spellings1 as the spelling list,

11. Otherwise DWIM attempts spelling correction using spellings2 as the spelling list.

If all fail, DWIM gives up.

17.5 DWIMUSERFN

Dwimuserfn provides a convenient way of adding to the transformations that DWIM performs, e.g., the user might want to change atoms of the form $X to (QA4LOOKUP X). The user defines dwimuserfn as a function of no arguments, and then enables it by setting dwimuserfn to T. DWIM will call dwimuserfn before attempting spelling correction, but after performing its other transformations, e.g. F/L, 8, 9, CLISP, etc. If dwimuserfn returns a non-NIL value, this value is treated as a form to be evaluated and returned as the value of faulteval or faultapply. Otherwise, if dwimuserfn returns NIL, DWIM proceeds as when dwimuserfn is not enabled, and attempts spelling correction. Note that in the event that dwimuserfn is to handle the correction, it is also responsible for any modifications to the original expression, i.e. DWIM simply takes its value and returns it.

In order for dwimuserfn to be able to function, it needs to know various things about the context of the error. Therefore, several of DWIM's internal variables have been made SPECVARS (See Section 18) and are therefore "visible" to dwimuserfn. Below are a list of those variables that may be useful.

faultx — for unbound atoms and undefined car of form, faultx is the atom or form. For undefined functions in apply, faultx is the name of the function.

faultargs — for undefined functions in apply, faultargs is the list of arguments.

faultapplyflg — Is T for undefined functions in apply. (Since faultargs may be NIL, faultapplyflg is necessary to distinguish between unbound atoms and undefined function in apply, since faultx is atomic in both cases).

tail — for unbound errors, tail is the tail car of which is the unbound atom. Thus dwimuserfn can replace the atom by another expression by performing (/RPLACA TAIL expr)

parent — for unbound atom errors, parent is the form in which the unbound atom appears, i.e. tail is a tail of parent.

type-in?	true if error occurred in type-in.
faultfn	name of function in which error occurred. (faultfn is TYPE-IN when the error occurred in type-in, and EVAL or APPLY when the error occurred under an explicit call to EVAL or APPLY).
dwimifyflg	true if error was encountered during dwimifying as opposed to during running the program.
expr	definition of faultfn, or argument to eval, i.e. the superform in which the error occurs.

17.6 Spelling Corrector Algorithm

The basic philosophy of DWIM spelling correction is to count the number of disagreements between two words, and use this number divided by the length of the longer of the two words as a measure of their relative disagreement. One minus this number is then the relative agreement or closeness. For example, CONS and CONX differ only in their last character. Such substitution errors count as one disagreement, so that the two words are in 75% agreement. Most calls to the spelling corrector specify rel=70,[24] so that a single substitution error is permitted in words of four characters or longer. However, spelling correction on shorter words is possible since certain types of differences such as single transpositions are not counted as disagreements. For example, AND and NAD have a relative agreement of 100.

The central function of the spelling corrector is chooz. chooz takes as arguments: a word, a spelling list, a minimum relative agreement, and an optional functional argument, xword, splst, rel, and fn respectively.[25]

---

[24] Integers between 0 and 100 are used instead of numbers between 0 and 1 in order to avoid floating point arithmetic.

[25] fn=NIL is equivalent to fn=(LAMBDA NIL T).

chooz proceeds down splst examining each word. Words not satisfying fn, or those obviously too long or too short to be sufficiently close to xword are immediately rejected. For example, if rel=70, and xword is 5 characters long, words longer than 7 characters will be rejected.[26]

If tword, the current word on splst, is not rejected, chooz computes the number of disagreements between it and xword by calling a subfunction, skor.

skor operates by scanning both words from left to right one character at a time.[27] Characters are considered to agree if they are the same characters; or appear on the same teletype key (i.e. a shift mistake), for example, * agrees with :, 1 with !,[28] etc.; or if the character in xword is a lower case version of the character in tword. Characters that agree are discarded, and the skoring continues on the rest of the characters in xword and tword.

If the first character in xword and tword do *not* agree, skor checks to see if either character is the same as one previously encountered, and not accounted-for at that time. (In other words, transpositions are not handled by lookahead, but by *lookback*.) A displacement of two or fewer positions is

---

[26] Special treatment is necessary for words shorter than xword, since doubled letters are not counted as disagreements. For example, CONNSSS and CONS have a relative agreement of 100. (Certain teletype diseases actually produce this sort of stuttering.) chooz handles this by counting the number of doubled characters in xword before it begins scanning splst, and taking this into account when deciding whether to reject shorter words.

[27] skor actually operates on the list of character codes for each word. This list is computed by chooz before calling skor using dchcon, so that no storage is used by the entire spelling correction process.

[28] For users on model 33 teletypes, as indicated by the value of model33flg being T, @ and P appear on the same key, as do L and /, N and L, and O and ←, and DWIM will proceed accordingly. The initial value for model33flg is T, except that various sites override this by use of the username-greeting feature described in Section 22.

counted as a tranposition; a displacement by more than two positions is counted as a disagreement. In either case, both characters are now considered as accounted for and are discarded, and skoring continues.

If the first character in xword and tword do not agree, and neither are equal to previously unaccounted-for characters, and tword has more characters remaining than xword, skor removes and saves the first character of tword, and continues by comparing the rest of tword with xword as described above. If tword has the same or fewer characters remaining than xword, the procedure is the same except that the character is removed from xword.[29] In this case, a special check is first made to see if that character is equal to the *previous* character in xword, or to the *next* character in xword, i.e. a double character typo, and if so, the character is considered accounted-for, and not counted as a disagreement.[30]

When skor has finished processing both xword and tword in this fashion, the value of skor is the number of unaccounted-for characters, plus the number of disagreements, plus the number of tranpositions, with two qualifications: (1) if both xword and tword have a character unaccounted-for in the same position, the two characters are counted only once, i.e. substitution errors count as only one disagreement, not two; and (2) if there are no unaccounted-for characters and no disagreements, transpositions are not counted. This permits spelling correction on very short words, such as edit commands, e.g.

---

[29] Whenever more than two characters in either xword or tword are unaccounted for, skoring is aborted, i.e. xword and tword are considered to disagree.

[30] In this case, the 'length' of xword is also decremented. Otherwise making xword sufficiently long by adding double characters would make it be arbitrarily close to tword, e.g. XXXXXX would correct to PP.

XRT->XTR.[31]

17.7 DWIM Functions

dwim[x] — If x=NIL, disables DWIM; value is NIL. If x=C, enables DWIM in cautious mode; value is CAUTIOUS. If x=T, enables DWIM in trusting mode; value is TRUSTING. For all other values of x, generates an error.

dwimify[x] — x is a form or the name of a function. dwimify performs all corrections and transformations that would occur if x were actually run. dwimify is undoable.

DW — edit macro. dwimifies current expression.

addspell[x;splst;n] — Adds x to one of the four spelling lists as follows:[32]

if splst=NIL, adds x to userwords and to spellings2. Used by defineq.

---

[31] Transpositions are also not counted when fastypeflg=T, for example, IPULX and IPLUS will be in 80% agreement with fastypeflg=T, only 60% with fastypeflg=NIL. The rationale behind this is that transpositions are much more common for fast typists, and should not be counted as disagreements, whereas more deliberate typists are not as likely to combine tranpositions and other mistakes in a single word, and therefore can use more conservative metric. fastypeflg is initially NIL.

[32] If x is already on the spelling list, and in its temporary section, addspell moves x to the front of that section. See page 17.13 for complete description of algorithm for maintaining spelling lists.

If splst=0, adds x to userwords. Used by load when loading exprs to property lists.

If splst=1, adds x to spellings1 (at end of permanent section). Used by lispx.

if splst=2, adds x to spellings2 (at end of permanent section). Used by lispx.

If splst=3, adds x to userwords and spellings3.

splst can also be a spelling list, in which case n is the (optional) length of the temporary section.

addspell sets lastword to x when splst=NIL, 0 or 3.

If x is not a literal atom, addspell takes no action.

misspelled?[xword;rel;splst;flg;tail;fn]

If xword=NIL or alt-mode, misspelled? prints = followed by the value of lastword, and returns this as the respelling, without asking for approval. Otherwise, misspelled? checks to see if xword is really misspelled, i.e. if fn applied to xword is true, or xword is already contained on splst. In this case, misspelled? simply returns xword. Otherwise misspelled? computes and returns fixspell[xword;rel;splst;flg;tail;fn].

fixspell[xword;rel;splst;flg;tail;fn][33]

The value of fixspell is either the respelling of xword or NIL. fixspell performs all of the interactions described earlier, including requesting user approval if necessary.

If xword=NIL or $ (alt-mode), the respelling is the value of lastword, and no approval is requested.

If flg=NIL, the correction is handled in type-in mode, i.e. approval is never requested, and xword is not typed. If flg=T, xword is typed (before the =) and approval is requested if approveflg=T.

If tail is not NIL, and the correction is successful, car of tail is replaced by the respelling (using /rplaca). In addition, fixspell will correct misspellings caused by running two words together.[34] In this case, car of tail is replaced by the two words, and the value of fixspell is the first one. For example, if fixspell is called to correct the edit command (MOVE TO AFTERCOND 3 2) with tail=(AFTERCOND 3 2), tail would be changed to (AFTER COND 2 3), and

---

[33] fixspell has some additional arguments, for internal use by DWIM.

[34] In this case, user approval is always requested. In addition, if the first word contains either fewer than 3 characters, or fewer characters than the second word, the default will be N.

fixspell would return AFTER (subject to user approval where necessary).[35]

fixspell generates an error if xword is already on splst.

The time required for a call to fixspell with a spelling list of length 60 when the entire list must be searched is .5 seconds. If fixspell determines that the first word on the spelling list is the respelling and does not need to search any further, the time required is .02 seconds. In other words, the time required is proportional to the number of words with which xword is compared, with the time for one comparison, i.e. one call skor, being roughly .01 seconds (varies slightly with the number of characters in the words being compared.)

The function chooz is provided for users desiring spelling correction without any output or interaction:

chooz[xword;rel;splst;tail;tieflg;fn][36] The value of chooz is the corrected spelling of xword[37] or NIL; chooz performs no interaction and no output. If tieflg=T and a tie occurs, i.e. more than one word on splst is found with the same closeness, chooz returns the first word. If tieflg=NIL, and a tie occurs, chooz returns NIL.

---

35 If tail=T, fixspell will also perform run-on corrections, returning a dotted pair of the two words in the event the correction is of this type.

36 chooz has some additional arguments, for internal use by DWIM.

37 chooz does not perform spelling *completion*, only spelling correction.

If tail is not NIL, chooz will correct a misspelling consisting of running two words together, e.g. (BREAKFOO) for (BREAK FOO). The value of chooz in this case is a dotted pair of the two words, e.g. (BREAK . FOO). Both DWIM and the editor use this option.

fncheck[fn;nomessflg;spellflg] The task of fncheck is to check whether fn is the name of a function and if not, to correct its spelling.[38] If fn is the name of a function or spelling correction is successful, fncheck adds the (corrected) name of the function to userwords using addspell, and returns it as its value.

nomessflg informs fncheck whether or not the calling function wants to handle the unsuccessful case: if nomessflg is T, fncheck simply returns NIL, otherwise it prints fn NOT A FUNCTION and generates a non-breaking error.

fncheck calls misspelled? to perform spelling correction, so that if fn=NIL, the value of lastword will be returned. spellflg corresponds to misspelled?'s fourth argument, flg. If spellflg=T, approval will be asked if DWIM was enabled in CAUTIOUS mode, i.e. if approveflg=T.

---

[38] Since fncheck is called by many low level functions such as arglist, unsavedef, etc., spelling correction only takes place when dwimflg=T, so that these functions can operate in a small INTERLISP system which does not contain DWIM.

fncheck is currently used by arglist, unsavedef, prettyprint, break0, breakin, chngnm, advise, printstructure, firstfn, lastfn, calls, and edita. For example, break0 calls fncheck with nomessflg=T since if fncheck cannot produce a function, break0 wants to define a dummy one. printstructure however calls fncheck with nomessflg=NIL, since it cannot operate without a function.

Many other system functions call misspelled? or fixspell directly. For example, breaki calls fixspell on unrecognized atomic inputs before attempting to evaluate them, using as a spelling list a list of all break commands. Similarly, lispx calls fixspell on atomic inputs using a list of all lispx commands. When unbreak is given the name of a function that is not broken, it calls fixspell with two different spelling lists, first with brokenfns, and if that fails, with userwords. makefile calls misspelled? using filelst as a spelling list. Finally, load, bcompl, brecompile, tcompl, and recompile all call misspelled? if their input file(s) won't open.

Index for Section 17

Page Numbers

SECTION 18

THE COMPILER AND ASSEMBLER[1]

18.1 The Compiler

The compiler is available in the standard INTERLISP system. It may be used to compile individual functions as requested or all function definitions in a standard format LOAD file. The resulting code may be stored as it is compiled, so as to be available for immediate use, or it may be written onto a file for subsequent loading. The compiler also provides a means of specifying sequences of machine instructions via ASSEMBLE.

The most common way to use the compiler is to compile from a symbolic (prettydef) file, producing a corresponding file which contains a set of functions in compiled form which can be quickly loaded. An alternate way of using the compiler is to compile from functions already defined in the user's INTERLISP system. In this case, the user has the option of specifying whether the code is to be saved on a file for subsequent loading, or the functions redefined, or both. In either case, the compiler will ask the user certain questions concerning the compilation. The first question is:

---

[1] The compiler itself, i.e. the part that actually generates code, was written and documented by, and is the responsibility of A.K. Hartley. The user interfaces, i.e. tcompl, recompile, bcompl, and brecompile, were written by W. Teitelman.

LISTING?

The answer to this question controls the generation of a listing and is explained in full below. However, for most applications, the user will want to answer this question with either ST or F, which will also specify an answer to the rest of the questions which would otherwise be asked. ST means the user wants the compiler to STore the new definitions; F means the user is only interested in compiling to a File, and no storing of definitions is performed. In both cases, the compiler will then ask the user one more question:

OUTPUT FILE:

to which the user can answer:

N or NIL	no output file.
File name	file is opened if not already opened, and compiled code is written on the file.

Example:

```
←COMPILE((FACT FACT1 FACT2))
LISTING? ST
OUTPUT FILE: FACT.COM
(FACT COMPILING)
     .
     .
(FACT REDEFINED)2
     .
     .
     .
(FACT2 REDEFINED)
(FACT FACT1 FACT2)
←
```

---

[2] compiler output and error messages are explained on page 18.48-52.

This process caused the functions FACT, FACT1, and FACT2 to be compiled, redefined, and the compiled definitions also written on the file FACT.COM for subsequent loading.

18.2 Compiler Questions

The compiler uses the free variables lapflg, strf, svflg, lcfil and lstfil which determines various modes of operation. These variables are set by the answers to the 'compset' questions. When any of the top level compiling functions are called, the function compset is called which asks a number of questions. Those that can be answered 'yes' or 'no' can be answered with YES, Y, or T for YES; and NO, N, or NIL for NO. The questions are:

1. LISTING?

The answer to this question controls the generation of a listing. Possible answers are:

- 1 Prints output of pass 1, the LAP macro code.[3]
- 2 Prints output of pass 2, the machine code.
- YES Prints output of both passes.
- NO Prints no listings.

The variable lapflg is set to the answer. If the answer is affirmative, compset will type FILE: to allow the user to indicate where the output is to be written. The variable lstfil is set to the answer.

---

[3] The LAP and machine code are usually not of interest but can be helpful in debugging macros.

There are three other possible answers to LISTING? - each of which specifies a complete mode for compiling. They are:

S   Same as last setting.

F   Compile to File (no definition of functions).

ST   STore new definitions.

Implicit in these three are the answers to the questions on disposition of compiled code and expr's, so questions 2 and 3 would not be asked if 1 were answered with S, F, or ST.

2. REDEFINE?

YES Causes each function to be redefined as it is compiled. The compiled code is stored and the function definition changed. The variable strf is set to T.

NO Causes function definitions to remain unchanged. The variable strf is set to NIL.

The answer ST for the first question implies YES for this question, F implies NO, and S makes no change.

3. SAVE EXPRS?

If answered YES, svflg is set to T, and the exprs are saved on the property list of the function name. Otherwise they are discarded. The answer ST for the first question implies YES for this question, F implies NO, and S makes no change.

4. OUTPUT FILE:

If the compiled definitions are to be written for later loading, you should provide the name of a file on which you wish to save the code that is generated. If you answer T or TTY:, the output will be typed on the teletype (not particularly useful). If you answer N, NO, or NIL, output will *not* be done. If the file named is already open, it will continue to be used. The free variable lcfil is set to the name of the file.

18.3 Nlambdas

When compiling the call to a function, the compiler must prepare the arguments to the function in one of three ways:

1. Evaluated (SUBR, SUBR*, EXPR, EXPR*, CEXPR, CEXPR*)
2. Unevaluated, spread (FSUBR, FEXPR, CFEXPR)
3. Unevaluated, not spread (FSUBR*, FEXPR*, CFEXPR*)

In attempting to determine which of these three is appropriate, the compiler will first look for a definition among the functions in the file that is being compiled. If the function is not contained there, the compiler will look for other information which can be supplied by the user by including nlambda nospread functions on the list nlama (for nlambda atoms), and including nlambda spread functions on the list nlaml (for nlambda list), and including lambda functions on the list lams.[4] If the function is not contained in the file,[5] or

---

[4] Including functions on lams is only necessary to override in-core nlambda definitions, since in the absence of other information, the compiler assumes the function is a lambda.

[5] The function can be defined anywhere in any of the files given as arguments to bcompl, tcompl, brecompile or recompile.

on the list nlama, nlaml, or lams, the compiler will look for a current definition. If the function is defined, its function type is assumed to be the desired type. If it is not defined, the compiler assumes that the function is of type 1, i.e. its arguments are to be evaluated.[6] [7] In other words, if there are type 2 or 3 functions called from the functions being compiled, and they are only defined in a separate file, they must be included on nlama or nlaml, or the compiler will incorrectly assume that their arguments are to be evaluated, and compile the calling function correspondingly. Note that this is only necessary if the compiler does not 'know' about the function. If the function is defined at compile time, or is handled via a macro, or is contained in the same group of files as the functions that call it, the compiler will automatically handle calls to that function correctly.

18.4 Globalvars

Another top level free variable that affects compilations is globalvars. Any variables that appear on the list globalvars, and are used freely in a compiled function, are always accessed through their value cell. In other words, a reference to the value of this variable is equivalent to (CAR (QUOTE variable)), regardless of whether or not it appears on the stack,

---

[6] Before making this assumption, if the value of compileuserfn is not NIL, the compiler calls (the value of) compileuserfn giving it as arguments cdr of the form and the form itself, i.e. the compiler does (APPLY* COMPILEUSERFN (CDR form) form). If a non-NIL value is returned, it is compiled instead of form. If NIL is returned, the compiler compiles the original expression as a call to a lambda-spread that is not yet defined. CLISP (Section 23) uses compileuserfn to tell the compiler how to compile iterative statements, IF-THEN-ELSE statements, and pattern match constructs.

[7] The names of functions so treated are added to the list alams (for assumed lamdas). alams is not used by the compiler; it is maintained for the user's benefit, i.e. so that the user can check to see whether any incorrect assumptions were made.

i.e., the stack is not even searched for this variable when the compiled function is entered. Similarly, (SETQ variable value) is equivalent to (RPLACA (QUOTE variable) value); i.e., it sets the top-level value.

All system parameters, unless otherwise specified, are GLOBALVARS, i.e. are members of the list globalvars, e.g. brokenfns, editmacros, #rpars, dwimflg, et al.[8] Thus, *rebinding* these variables will not affect the behavior of the system: instead, the variables must be *reset* to their new values, and if they are to be restored to their original values, reset again. For example, the user might write ...(SETQ globalvar new-value) form (SETQ globalvar old-value). Note that in this case, if an error occurred during the evaluation of form, or a control-D was typed, the global variable would not be restored to its original value. The function resetvar (described in Section 5) provides a convenient way of resetting global variables in such a way that their values are restored even if an error occurred or control-D is typed.

18.5 Compiler Functions

Note: when a function is compiled from its in core definition, i.e., via compile (and certain calls to recompile), as opposed to tcompl (which uses the definitions on a file), and the function has been modified by break, trace, breakin, or advise, it is restored to its original state, and a message printed out, e.g., FOO UNBROKEN. Then, if the function is not defined as an expr, its property list is searched for the property EXPR (see savedef, Section 8). If there is a property EXPR, its value is used for the compilation, otherwise, the compiler prints (fn NOT COMPILEABLE), and goes on to the next function.

---

[8] Since the stack does not have to be searched to find the values of these variables, a considerable savings in time is achieved, especially for deep computations.

compile[x;flg]

x is a list of functions (if atomic, list[x] is used). compile first asks the standard compiler questions, and then compiles each function on x, using its in-core definition. Value is x.

If compiled definitions are being dumped to a file, the file is closed unless flg=T.

compile1[name;def]

compiles def, redefining name if strf=T.[9] compile1 is used by compile, tcompl, and recompile. If dwimifycompflg is T, or def contains a CLISP declaration, def is dwimified before compiling. See Section 23.

tcompl[files]

tcompl is used to 'compile files', i.e., given a symbolic load file (e.g., one created by prettydef), it produces a file that contains the same S-expressions as the original symbolic file, except that every defineq is replaced by the corresponding compiled definitions. This 'compiled' file can be loaded into any INTERLISP system with load.

files is a list of symbolic files to be compiled (if atomic, list[files] is used). tcompl asks the standard compiler questions, except for OUTPUT FILE: Instead, the output from the compilation of each symbolic file is written on a

---

[9] strf is one of the variables set by compset, described earlier.

file of the same name suffixed with COM, e.g., tcompl[(SYM1 SYM2)] produces two files, SYM1.COM and SYM2.COM.[10]

tcompl processes each file one at a time, reading in the entire file. Then, for each DEFINEQ, tcompl adds any NLAMBDA's in the DEFINEQ to nlama or nlaml,[11] and adds lambdas to the list lams,[12] so that calls to these functions will be compiled correctly. Expressions beginning with DECLARE can be used to affect the compilation, e.g. set up MACROS. tcompl evaluates each expression in (cdr of) the DECLARE, presumably for effect.[13] tcompl then compiles each function in the DEFINEQ's. Finally, all other expressions in the file, e.g. RPAQQ's, DEFLIST's, etc., are written onto the output file.[14]

---

[10] The file name is constructed from the name field only, e.g. tcompl[<BOBROW>FOO.TEM;3] produces FOO.COM on the connected directory. The version number will be the standard default.

[11] described earlier, page 18.5.

[12] nlama, nlaml, and lams are rebound to their top level values (using resetvar) by tcompl, recompile, bcompl, brecompile, compile, and blockcompile, so that any additions to these lists while inside of these functions will not propagate outside.

[13] DECLARE is *defined* the same as QUOTE, so it will have no effect when the symbolic file is loaded. tcompl, recompile, bcompl, and brecompile also evaluate any DEFLIST expression that was output by a COMPROP or COMPROP* prettydef command (see Section 14).

[14] except for DEFLISTs output by a COMPROP* prettydef command.

The value of tcompl is a list of the names of the output files. All files are properly terminated and closed.

Recompile

The purpose of recompile is to allow the user to update a compiled file without recompiling every function in the file. Recompile does this by using the results of a previous compilation. It produces a compiled file similar to one that would have been produced by tcompl, but at a considerable savings in time by compiling selected functions and copying from an earlier tcompl or recompile file the compiled definitions for the remainder of the functions in the file. Even more savings can be achieved if the symbolic file being recompiled is currently in-core, i.e., was previously loaded, or was made from the user's current system. In this case, recompile will not have to read in the file, but can work from the in-core definitions.[15]

If the functions to be recompiled are currently defined as exprs, then recompile can be called with just one argument, the symbolic file; the rest of the arguments will be set appropriately. In other words, the most common usage of recompile is in the following sequence, load[file;PROP], edit some functions (thus unsavedefing them), makefile[file], and recompile[file], producing a new compiled file exactly equivalent to tcompl[file]. The rest of the discussion of recompile explains nonstandard usages, e.g., the symbolic file has not been loaded, some of the functions that have been changed are currently not unsaved, etc.

---

[15] This requires that the user observe the conventions of the 'file package' described in Section 14 when making the symbolic file, i.e., he used makefile or else used prettydef with arguments of the form fileFNS, file, and fileVARS.

recompile[pfile;cfile;fns;coreflg] pfile is the name of the pretty file to be compiled, cfile is the name of the compiled file containing compiled definitions that may be copied. fns is a list of the functions in pfile that are to be recompiled, i.e., they have been changed (or defined for the first time) since cfile was made. Note that pfile, not fns, drives recompile, so that extra functions may appear on fns. If fns=T, all function in pfile currently defined as exprs (after unbreaking and unadvising) are recompiled.

recompile asks the standard compiler questions, except for OUTPUT FILE: As with tcompl, the output automatically goes to pfile.COM.[16] recompile then reads in pfile. As with tcompl, for each DEFINEQ, the NLAMBDAs are added to nlama and nlaml, and LAMBDAs are added to lams. Similarly, DECLAREs and DEFLISTs are treated the same as with tcompl. Then each function is compiled if it appears on fns, or fns=T and the function is an expr. Otherwise, recompile reads from cfile until it finds the compiled version of the function it is working on, and then copies it (and all compiler generated subfunctions) to pfile.COM. Finally, all other expressions are written onto pfile.COM.

---

16 In general, all constructions of the form pfile.COM, pfileFNS, and pfileVARS are performed using the name field only. For example, if pfile=<BOBROW>FOO.TEM;3, pfile.COM means FOO.COM, pfileFNS means FOOFNS, etc.

Note that the user can thus modify an old compiled file so as to add new functions by prettydefing them in pfile and then including them on fns. Similarly, he can delete functions by simply not prettydefing them, since if they do not appear in pfile, they will never be compiled or copied to pfile.COM. Note, however that the entire process depends on the order of those functions in cfile that are to be copied being the same as those in pfile. For example, if FOO appears before FIE in cfile, but the order is reversed in pfile, then when recompile attempts to copy FIE, it will skip over FOO. Then when it attempts to copy FOO, it will read to the end of cfile and not find it. In this case, it will generate an error FOO NOT FOUND.

If the file pfile is in core, i.e., has been loaded, or else was prettydefed from this system, the user can take advantage of this by calling recompile with coreflg=T. In this case, the procedure is the same as described above, but recompile 'fakes' reading pfile, instead determining what is on pfile from pfilefns and pfilevars[17] (recompile does read the date from pfile, which it copies to the output file.)

recompile will work correctly even for functions written via the third argument to prettydef using a FNS command. (See Section 14).

If cfile=NIL, pfile.COM is used for copying

---

[17] See footnote on page 18.11.

from.[18] In addition, if both fns and coreflg are NIL, they are set to T. This is the most common usage.

The value of recompile is the new compiled file, pfile.COM.

18.6 Open Functions

When a function is called from a compiled function, a system routine is invoked that sets up the parameter and control push lists as necessary for variable bindings and return information. As a result, function calls can take up to 350 microseconds per call. If the amount of time spent *inside* the function is small, this function calling time will be a significant percentage of the total time required to use the function. Therefore, many 'small' functions, e.g., car, cdr, eq, not, cons are always compiled 'open', i.e., they do not result in a function call. Other larger functions such as prog, selectq, mapc, etc. are compiled open because they are frequently used. It is useful to know exactly which functions are compiled open in order to determine where a program is spending its time. Therefore below is a list of those functions which when compiled do not result in function calls. Note that the next section tells how the user can make other functions compile open via MACRO definitions.[19]

---

[18] In other words, if cfile, the file used for obtaining compiled definitions to be copied, is NIL, pfile.COM is used, i.e., same name as output but a different version number (one less) than the output file.

[19] The user can also affect the compiled code via compileuserfn, described in footnote on page 18.6.

The following functions compile open:

AC, ADD1, AND, APPLY*, ARG, ARRAYP, ASSEMBLE, ATOM, BLKAPPLY, BLKAPPLY*, CAR, CDR, CAAR, ... CDDDAR, CDDDDR, CLOSER, COND, CONS, EQ, ERSETQ, EVERY, EVQ, FASSOC, FCHARACTER, FDIFFERENCE, FGTP, FIX, FIXP, FLAST, FLENGTH, FLOAT, FLOATP, FMEMB, FMINUS, FNTH, FPLUS, FQUOTIENT, FRPLACA, FRPLACD, FSTKARG, FSTKNTH, FTIMES, FUNCTION, GETHASH, GO, IDIFFERENCE, IGREATERP, ILESSP, IMINUS, IPLUS, IQUOTIENT, IREMAINDER, ITIMES, LIST, LISTP, LITATOM, LLSH, LOC, LOGAND, LOGOR, LOGXOR, LRSH, LSH, MAP, MAPC, MAPCAR, MAPCON, MAPCONC, MAPLIST, MINUSP, NEQ, NLISTP, NLSETQ, NOT, NOTEVERY, NOTANY, NULL, NUMBERP, OPENR, OR, PROG, PROG1, PROGN, RESETFORM, RESETVAR, RETURN, RPTQ, RSH, SELECTQ, SETARG, SETN, SETQ, SMALLP, SOME, STRINGP, SUB1, SUBSET, UNDONLSETQ, VAG, ZEROP

18.7 Compiler Macros

The INTERLISP compiler includes a macro capability by which the user can affect the compiled code. Macros are defined by placing the macro definition on the property list of the corresponding function under the property MACRO.[20] When the compiler begins compiling a form, it retrieves a macro definition for car of the form, if any, and uses it to direct the compilation.[21] The three different types of macro definitions are given below.

---

[20] An expression of the form (DECLARE (DEFLIST ... (QUOTE MACRO))) can be used *within* a function to define a MACRO. DECLARE is defined the same as QUOTE and thus can be placed so as to have no effect on the running of the function.

[21] The compiler has built into it how to compile certain basic functions such as car, prog, etc., so that these will not be affected by macro definitions. These functions are listed above. However, some of them are themselves implemented via macros, so that the user could change the way they compile.

(1) Open macros - (LAMBDA ...) or (NLAMBDA ...)

A function can be made to compile open by giving it a macro definition of the form (LAMBDA ...) or (NLAMBDA ...), e.g., (LAMBDA (X) (COND ((GREATERP X 0) X) (T (MINUS X)))) for abs. The effect is the same as though the macro definition were written in place of the function wherever it appears in a function being compiled, i.e., it compiles as an open LAMBDA or NLAMBDA expression. This saves the time necessary to call the function at the price of more compiled code generated.

(2) Computed macros - (atom expression)

A macro definition beginning with an atom other than LAMBDA, NLAMBDA, or NIL, allows *computation* of the INTERLISP expression that is to be compiled in place of the form. The atom which starts the macro definition is bound to cdr of the form being compiled. The expression following the atom is then evaluated, and the result of this evaluation is compiled in place of the form. For example, list could be compiled this way by giving it the macro definition:

```
[X (LIST (QUOTE CONS)
         (CAR X)
         (AND (CDR X)
              (CONS (QUOTE LIST)
                    (CDR X]
```

This would cause (LIST X Y Z) to compile as (CONS X (CONS Y (CONS Z NIL))). Note the recursion in the macro expansion.[22] Ersetq, nlsetq, map, mapc, mapcar, mapconc, and some, are compiled via macro definitions of this type.

If the result of the evaluation is the atom INSTRUCTIONS, no code will be generated by the compiler. It is then assumed the evaluation was done for effect and the necessary code, if any, has been added. This is a way of giving direct instructions to the compiler if you understand it.

---

[22] list is actually compiled more efficiently.

(3) Substitution macro - (NIL expression) or (list expression)

Each argument in the form being compiled is substituted for the corresponding atom in car of the macro definition, and the result of the substitution is compiled instead of the form, i.e., (SUBPAIR (CAR macrodef) (CDR form) (CADR macrodef)). For example, the macro definition of add1 is ((X) (IPLUS X 1)). Thus, (ADD1 (CAR Y)) is compiled as (IPLUS (CAR Y) 1). The functions add1, sub1, neq, nlistp, zerop, flength, fmemb, fassoc, flast, and fnth are all compiled open using substitution macros. Note that abs could be compiled open as shown earlier or via a substitution macro. A substitution macro, however, would cause (ABS (FOO X)) to compile as (COND ((GREATERP (FOO X) 0) (FOO X)) (T (MINUS (FOO X)))) and consequently (FOO X) would be evaluated three times.

18.8 FUNCTION and Functional Arguments

Expressions that begin with FUNCTION will always be compiled as separate functions[23] named by attaching a gensym to the end of the name of the function in which they appear, e.g., FOOA0003.[24] This gensym function will be called at run time. Thus if FOO is defined as (LAMBDA (X) ... (FOO1 X (FUNCTION ...)) ...) and compiled, then when FOO is run, FOO1 will be called with two arguments, X, and FOOA000n,[25] and then FOO1 will call FOOA000n each time it must use its functional argument.

---

[23] except when they are compiled open, as is the case with most of the mapping functions.

[24] nlsetq and ersetq expressions also compile using gensym functions. As a result, a go or return cannot be used inside of a compiled nlsetq or ersetq if the corresponding prog is outside, i.e. above the nlsetq or ersetq.

[25] or an appropriate funarg expression, see Section 11.

Note that a considerable savings in time could be achieved by defining FOO1 as a computed macro of the form:

```
(Z (LIST (SUBST (CADADR Z) (QUOTE FN) def) (CAR Z)))
```

where def is the definition of FOO1 as a function of just its first argument and FN is the name used for its functional argument in its definition. The expression compiled contains what was previously the functional argument to FOO1, as an open LAMBDA expression. Thus you save not only the function call to FOO1, but also each of the function calls to its functional argument. For example, if FOO1 operates on a list of length ten, eleven function calls will be saved. Of course, this savings in time cost space, and the user must decide which is more important.

18.9 Block Compiling

Block compiling provides a way of compiling several functions into a single block. Function calls between the component functions of the block are very fast, and the price of using a free variable, namely the time required to look up its value on the stack, is paid only once - when the block is entered. Thus, compiling a block consisting of just a single recursive function may be yield great savings if the function calls itself many times, e.g., equal, copy, and count are block compiled in INTERLISP.

The output of a block compilation is a single, usually large, function. This function looks like any other compiled function; it can be broken, advised, printstructured, etc. Calls from within the block to functions outside of the block look like regular function calls, except that they are usually linked (described below). A block can be entered via several different functions,

called entries. These must be specified when the block is compiled.[26] For example, the error block has three entries, errorx, interrupt, and fault1. Similarly, the compiler block has nine entries.

Specvars

One savings in block compiled functions results from not having to store on the stack the names of the variables bound within the block, since the block functions all 'know' where the variables are stored. However, if a variable bound in a block is to be referenced *outside* the block, it must be included on the list specvars.[27] For example, helpclock is on specvars, since it is rebound inside of lispxblock and editblock, but the error functions must be able to obtain its latest value.

Localfreevars

Localfreevars is a feature designed for those variables which are used freely by one or more of the block functions, but which are always bound (by some other block function) before they are referenced, i.e. their free values above the block are never used. Normally, when a block is entered, all variables which are used freely by any function in the block are looked up and pointers to the bindings are stored on the stack. When any of these variables are

---

[26] Actually the block is entered the same as every other function, i.e., at the top. However, the entry functions call the main block with their name as one of its arguments, and the block dispatches on the name, and jumps to the portion of the block corresponding to that entry point. The effect is thus the same as though there were several different entry points.

[27] Arguments to the block that are referenced freely outside the block must also be SPECVARS if they are reset within the block, or else the new value will not be obtained.

rebound in the block, the old pointer is saved and a pointer to the new binding is stored in the original stack position. It frequently happens that variables used freely within a block are in fact always bound within the block prior to the free reference. The unnecessary lookup of the value of the free variable at the time of entry to the block can be avoided by putting the variable name on the list localfreevars. If a variable is on localfreevars, its value will *not* be looked up at the time of entry. When the variable is bound, the value will be stored in the proper stack position. Should the variable in fact be referenced before it is bound, the program will still work correctly. Invisible to the user, a rather time-comsuming process will take place. The reference will cause a trap which will invoke code to determine which variable was referenced and look up the value. Future references to that variable during this call to the block will be normal, i.e. will not cause a trap.

trapcount[x] — is a function to monitor the performance of block compiled code with respect to localfreevars. If x is NIL, trapcount returns the cumulative number of traps caused by localfreevars that were not bound before use. If x is a number, the trapcount is reset to that number.

evq is another compiler artifice for free variables references. (EVQ X) has the effect of (EVAL (QUOTE X)) without the call to eval (if X is an atom). evq is intended primarily for use in conjunction with localfreevars. For example, suppose a block consists of three functions, FOO1, FOO2, and FOO3, with FOO1 and FOO2 being entries, and FOO3 using X freely, where X is bound in FOO1, but not in FOO2, i.e. FOO1 rebinds X, but when entered via FOO2, the user intends X to be used freely, and its higher value obtained. If X is on localfreevars, then each time the block is entered via FOO2, a trap will occur when FOO3 first references X. In order to avoid this, the user can insert (EVQ X) in FOO2. This will circumvent the trap by explicitly invoking the routine that searches

back up the stack for the last binding of X. Thus, when used with localfreevars, evq does two things: it returns the value of its argument, and also stores that value in the binding slot for the variable so that no future references to that variable (in this call) will cause traps. Since the time consumed by the trap can greatly exceed the time required for a variable lookup, using evq in these situations can result in a considerable savings.

Retfns

Another savings in block compilation arises from omitting most of the information on the stack about internal calls between functions in the block. However, if a function's name must be visible on the stack, e.g., if the function is to be returned from retfrom, it must be included on the list retfns.

Blkapplyfns

Normally, a call to apply from inside a block would be the same as a call to any other function outside of the block. If the first argument to apply turned out to be one of the entries to the block, the block would have to be reentered. blkapplyfns enables a program to compute the name of a function in the block to be called next, without the overhead of leaving the block and reentering it. This is done by including on the list blkapplyfns those functions which will be called in this fashion, and by using blkapply in place of apply, and blkapply* in place of apply*. For example, the calls to the functions handling RI, RO, LI, LO, BI, and BO in the editor are handled this way. If blkapply or blkapply* is given a function not on blkapplyfns, the effect is the same as a call to apply or apply* and no error is generated. Note however, that blkapplyfns must be set at *compile* time, not run time, and furthermore, that all functions on blkapplyfns must be in the block, or an error is generated (at compile time), NOT ON BLKFNS.

Blklibrary

Compiling a function open via a macro provides a way of eliminating a function call. For block compiling, the same effect can be achieved by including the function in the block. A further advantage is that the code for this function will appear only once in the block, whereas when a function is compiled open, its code appears at each place where it is called.

The block library feature provides a convenient way of including functions in a block. It is just a convenience since the user can always achieve the same effect by specifying the function(s) in question as one of the block functions, provided it has an expr definition at compile time. The block library feature simply eliminates the burden of supplying this definition.

To use the block library feature, place the names of the functions of interest on the list blklibrary, and their EXPR definition on the property list of the function under the property BLKLIBRARYDEF. When the block compiler compiles a form, it first check to see if the function being called is one of the block functions. If not, and the function is on blklibrary, its definition is obtained from the property value of BLKLIBRARYDEF, and it is automatically included as part of the block. The functions assoc, equal, getp, last, length, lispxmatch, memb, nconc1, nleft, nth, and /rplnode already have BLKLIBRARYDEF properties.

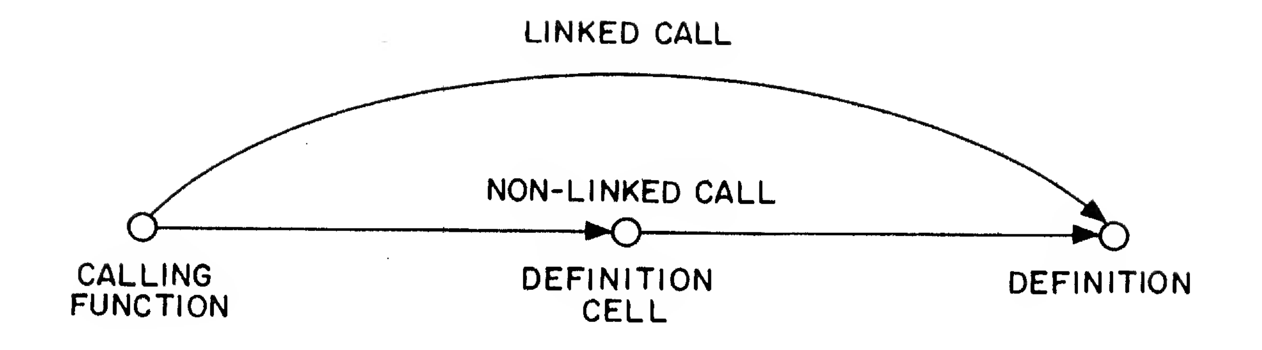
18.10 Linked Function Calls

Conventional (non-linked) function calls from a compiled function go through the function definition cell, i.e., the definition of the called function is obtained from its function definition cell at call time. Thus, when the user breaks, advises, or otherwise modifies the definition of the function FOO,

every function that subsequently calls it instead calls the modified function. For calls from the system functions, this is clearly *not* a feature. For example, the user may wish to break on basic functions such as print, eval, rplaca, etc., which are used by the break package. In other words, we would like to guarantee that the system packages will survive through user modification (or destruction) of basic functions (unless the user specifically requests that the system packages also be modified). This protection is achieved by linked function calls.

For linked function calls, the definition of the called function is obtained at *link* time, i.e., when the calling function is defined, and stored in the literal table of the calling function. At *call* time, this definition is retrieved from where it was stored in the literal table, *not* from the function definition cell of the called function as it is for non-linked calls. These two different types of calls are illustrated in Figure 18-1.

Note that while function calls from block compiled functions are *usually* linked, and those from standardly compiled functions are *usually* non-linked, linking function calls and blockcompiling are independent features of the INTERLISP compiler, i.e., linked function calls are possible, and frequently employed, from standardly compiled functions.


LINKED CALL
NON-LINKED CALL
CALLING
FUNCTION
DEFINITION
CELL
DEFINITION

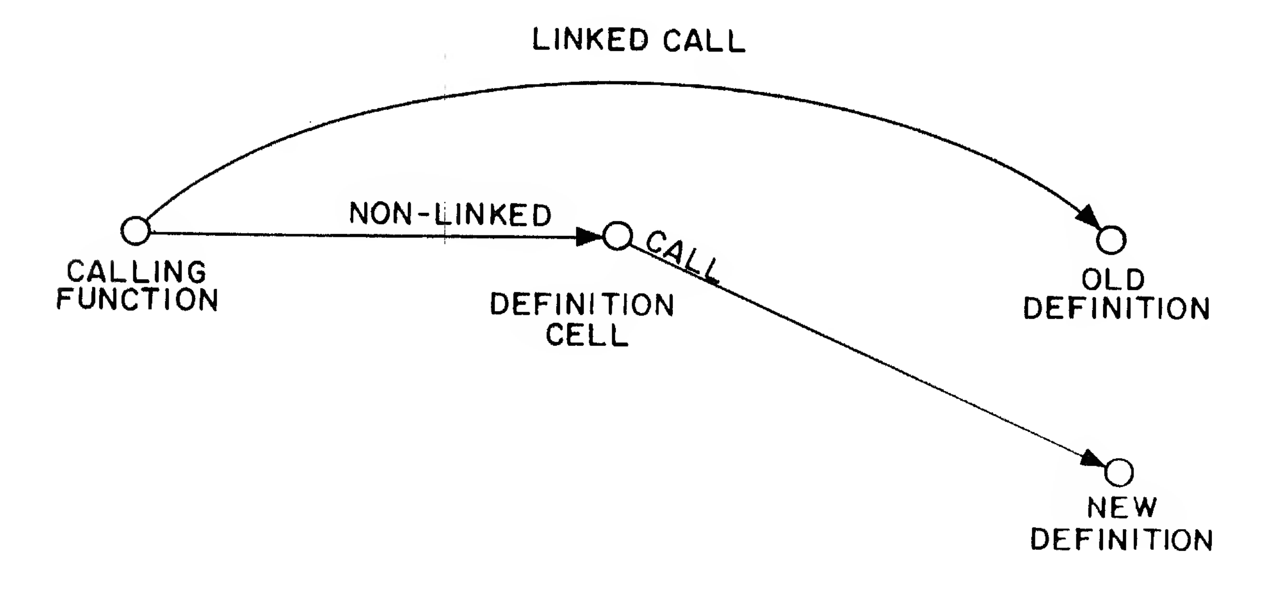

LINKED CALL
NON-LINKED
CALL
CALLING
FUNCTION
DEFINITION
CELL
OLD
DEFINITION
NEW
DEFINITION

FIGURE 18-1

Note that normal function calls require only the called function's name in the literals of the compiled code, whereas a linked function call uses two literals and hence produces slightly larger compiled functions.

The compiler's decision as to whether to link a particular function call is determined by the variables linkfns and nolinkfns as follows:

(1) If the function appears on nolinkfns, the call is not linked;

(2) If block compiling and the function is one of the block functions, the call is internal as described earlier;

(3) If the function appears on linkfns, the call is linked;

(4) If nolinkfns=T, the call is not linked;

(5) If block compiling, the call is linked;

(6) If linkfns=T, the call is linked;

(7) Otherwise the call is not linked.

Note that (1) takes precedence over (2), i.e., if a function appears on nolinkfns, the call to it is *not* linked, even if it is one of the functions in the block, i.e., the call will go outside of the block.

Nolinkfns is initialized to various system functions such as errorset, break1, etc. Linkfns is initialized to NIL. Thus if the user does not specify otherwise, all calls from a block compiled function (except for those to functions on nolinkfns) will be linked; all calls from standardly compiled functions will not be linked. However, when compiling system functions such as help, error, arglist, fntyp, break1, et al, linkfns is set to T so that even though these functions are not block compiled, all of their calls will be linked.

If a function is not defined at link time, i.e., when an attempt is made to link to it, a message is printed, fn1 NOT DEFINED WHEN LINK TRIED FROM fn2.

When the function is later defined, the link can be completed by relinking the calling function using relink described below. Otherwise, if a function is run which attempts a linked call that was not completed, faultapply is called. If the function is now defined, i.e., it was defined at some point after the attempt was made to link to it, faultapply will quietly perform the link and continue the call. Otherwise, it will print U.D.F. and proceed as described in Section 16.

Linked function calls are printed on the backtrace as ;fn; where fn is the name of the function. Note that this name does *not* actually appear on the stack, and that stkpos, retfrom, and the rest of the pushdown list functions (Section 12) will *not* be able to find it. Functions which must be visible on the stack should not be linked to, i.e., include them on nolinkfns when compiling a function that would otherwise link its calls.

printstructure, calls, break on fn1-IN-fn2 and advise fn1-IN-fn2 all work correctly for linked functions calls, e.g., break[(FOO IN FIE)], where FOO is called from FIE via a linked function call.

Relinking

The function relink is available for relinking a compiled function, i.e., updating all of its linked calls so that they use the definition extant at the time of the relink operation.

relink[fn] — fn is either WORLD, the name of a function, a list of functions, or an atom whose value is a list of functions. relink performs the corresponding relinking operations. relink[WORLD] is possible because laprd maintains on linkedfns a list of all

user functions containing any linked calls. syslinkedfns is a list of all system functions that have any linked calls. relink[WORLD] performs both relink[linkedfns] and relink[syslinkedfns].

The value of relink is fn.

It is important to stress that linking takes place when a function is *defined*. Thus, if FOO calls FIE via a linked call, and a bug is found in FIE, changing FIE is not sufficient; FOO must be relinked. Similarly, if FOO1, FOO2, and FOO3 are defined (in that order) in a file, and each call the others via linked calls, when a new version of the file is loaded, FOO1 will be linked to the *old* FOO2 and FOO3, since those definitions will be extant at the time it is read and defined. Similarly, FOO2 will link to the new FOO1 and *old* FOO3. Only FOO3 will link to the new FOO1 and FOO2. The user would have to perform relink[FOOFNS] following the load.

18.11 The Block Compiler

There are three user level functions for blockcompiling, blockcompile, bcompl, and brecompile, corresponding to compile, tcompl, and recompile. All of them ultimately call the same low level functions in the compiler, i.e., there is no 'blockcompiler' per se. Instead, when blockcompiling, a flag is set to enable special treatment for specvars, retfns, blkapplyfns, and for determining whether or not to link a function call. Note that all of the previous remarks on macros, globalvars, compiler messages, etc., all apply equally for block compiling. Using block declarations described below, the user can intermix in a single file functions compiled normally, functions compiled normally with linked calls, and block compiled functions.

Blockcompile

blockcompile[blkname;blkfns;entries;flg] blkfns is a list of the functions comprising the block, blkname is the name of the block, entries a list of entries to the block, e.g.,

←BLOCKCOMPILE(SUBPRBLOCK (SUBPAIR SUBLIS SUBPR) (SUBPAIR SUBLIS))

Each of the entries must also be on blkfns or an error is generated, NOT ON BLKFNS.[28]

If entries is NIL, list[blkname] is used, e.g.,

←BLOCKCOMPILE(COUNT (COUNT COUNT1))

If blkfns is NIL, list[blkname] is used, e.g.,

←BLOCKCOMPILE(EQUAL)

blockcompile asks the standard compiler questions and then begins compiling. As with compile, if the compiled code is being written to a file, the file is closed unless flg=T. The value of blockcompile is a list of the entries, or if entries=NIL, the value is blkname.

The output of a call to blockcompile is one

---

[28] If only one entry is specified, the block name can also be one of the blkfns, e.g. BLOCKCOMPILE(FOO (FOO FIE FUM) (FOO)). However, if more than one entry is specified, an error will be generated, CAN'T BE BOTH AN ENTRY AND THE BLOCK NAME.

function definition for blkname, plus definitions for each of the functions on entries if any. These entry functions are very short functions which immediately call blkname.

Block Declarations

Since block compiling a file frequently involves giving the compiler a lot of information about the nature and structure of the compilation, e.g., block functions, entries, specvars, linking, et al, we have implemented a special prettydef command to facilitate this commmunication. The user includes in the third argument to prettydef a command of the form (BLOCKS $block_1$ ... $block_2$ ... $block_n$) where each $block_i$ is a block declaration. bcompl and brecompile described below are sensitive to these declarations and take the appropriate action.

The form of a block declaration is:

(blkname $blkfn_1$ ... $blkfn_m$ (var_1 . value) ... (var_n . value))

$blkfn_1$ ... $blkfn_m$ are the functions in the block and correspond to blkfns in the call to blockcompile. The (var . value) expressions indicate the settings for variables affecting the compilation.

As an example, the value of editblocks is shown below. It consists of three block declarations, editblock, editfindblock, and edit4e.

```
[RPAQQ EDITBLOCKS
      ((EDITBLOCK EDITL0 EDITL1 UNDOEDITL EDITCOM EDITCOMA EDITCOML
                  EDITMAC EDITCOMS EDIT]UNDO UNDOEDITCOM
                  UNDOEDITCOM1 EDITSMASH EDITNCONC EDIT1F EDIT2F
                  EDITNTH BPNT BPNT0 BPNT1 RI RO LI LO BI BO
                  EDITDEFAULT ## EDUP EDIT* EDOR EDRPT EDLOC EDLOCL
                  EDIT: EDITMBD EDITXTR EDITELT EDITCONT EDITSW
                  EDITMV EDITTO EDITBELOW EDITRAN TAILP EDITSAVE
                  EDITH (ENTRIES EDITL0 ## UNDOEDITL)
                  (SPECVARS L COM LCFLG #1 #2 #3 LISPXBUFS
                            **COMMENT**FLG PRETTYFLG UNDOLST
                            UNDOLST1)
                  (RETFNS EDITL0)
                  (GLOBALVARS EDITCOMSA EDITCOMSL EDITOPS
                             HISTORYCOMS EDITRACEFN)
                  (BLKAPPLYFNS RI RO LI LO BI BO EDIT: EDITMBD
                              EDITMV EDITXTR)
                  (BLKLIBRARY LENGTH NTH LAST)
                  (NOLINKFNS EDITRACEFN))
       (EDITFINDBLOCK EDIT4E EDIT4E1 EDITQF EDIT4F EDITFPAT
                      EDITFPAT1 EDIT4F1 EDIT4F2 EDIT4F3 EDITSMASH
                      EDITFINDP EDITBF EDITBF1 ESUBST
                      (ENTRIES EDITQF EDIT4F EDITFPAT EDITFINDP
                               EDITBF ESUBST))
       (EDIT4EBLOCK EDIT4E EDIT4E1 (ENTRIES EDIT4E EDIT4E1]
```

Whenever bcompl or brecompile encounter a block declaraction[29] they rebind retfns, specvars, localfreevars, globalvars, blklibrary, nolinkfns, and linkfns to their top level value, bind blkapplyfns and entries to NIL, and bind blkname to the first element of the declaration. They then scan the rest of the declaration, gathering up all atoms, and setting car of each nonatomic element to cdr of the expression if atomic, e.g., (LINKFNS . T), or else to union of cdr of the expressions with the current (rebound) value,[30] e.g., (GLOBALVARS EDITCOMSA EDITCOMSL). When the declaration is exhausted, the block compiler is called and given blkname, the list of block functions, and entries.

Note that since all compiler variables are rebound for each block declaration,

---

[29] The BLOCKS command outputs a DECLARE expression, which is noticed by bcompl and brecompile.

[30] Expressions of the form (var * form) will cause form to be evaluated and the resulting list used as described above, e.g. (GLOBALVARS * MYGLOBALVARS).

the declaration only has to set those variables it wants *changed*. Furthermore, setting a variable in one declaration has no effect on the variable's value for another declaration.

After finishing all blocks, bcompl and brecompile treat any functions in the file that did not appear in a block declaration in the same way as do tcompl and recompile. If the user wishes a function compiled separately as well as in a block, or if he wishes to compile some functions (not blockcompile), with some compiler variables changed, he can use a special pseudo-block declaration of the form (NIL fn_1 ... fn_m (var_1 . value) ... (var_n . value)) which means compile fn_1 ... fn_m after first setting var_1 ... var_n as described above. For example, (NIL CGETD FNTYP ARGLIST NARGS NCONC1 GENSYM (LINKFNS . T)) appearing as a 'block declaration' will cause the six indicated functions to be compiled while linkfns=T so that all of their calls will be linked (except for those functions on nolinkfns).

bcompl

bcompl[files;cfile]

files is a list of prettydefed files. (If atomic, list[files] is used.) bcompl differs from tcompl in that it compiles all of the files at once, instead of one at a time. This is to permit one block to contain functions in several files.[31] Output is to cfile if given, otherwise to a file whose name is car[files] suffixed with COM[32] e.g., bcompl[(EDIT WEDIT)] produces one file, EDIT.COM.

---

[31] Thus if you have several files to be bcompled *separately*, you must make several calls to bcompl.

[32] See footnote on page 18.11.

bcompl asks the standard compiler questions, except for OUTPUT FILE: then reads in all of the files, adds all nlambda functions in DEFINEQ's to nlama, nlaml, lambdas to lams, evaluates DECLARE expressions,[33] and then processes the block declarations as described above. Finally, it compiles any functions not mentioned in one of the declarations and writes out all other expressions.

The value of bcompl is the output file.

Note that it is permissible to tcompl files set up for bcompl; the block declarations will simply have no effect. Similarly, you can bcompl a file that does not contain any block declarations and the result will be the same as having tcompled it.

Brecompile

The purpose of brecompile is to allow the user to update a compiled file without requiring an entire bcompl. As with recompile, the usual way to call brecompile involves specifying just its first argument, the symbolic file(s), as in the sequence of loading file(s) to PROP, editing selected definitions, makefiling, and then calling brecompile. In this case, brecompile recompiles all exprs and works from in-core definitions.[34] Note that this assumes that

---

[33] See footnote on page 18.9.

[34] Note that if any of the functions in a block are recompiled, the entire block is recompiled.

each symbolic file was produced by makefile, i.e., the arguments to prettydef were fileFNS, file, and fileVARS, since brecompile uses fileFNS and file VARS to drive its operation. The rest of the discussion below is for various nonstandard usages.

brecompile[files;cfile;fns;coreflg] files is a list of symbolic files (if atomic, list[files] is used). cfile is the compiled file corresponding to bcompl[files] or a previous brecompile, i.e., it contains compiled definitions that may be copied.

fns is a list of those functions to be recompiled, i.e., they have been changed (or defined for the first time) since cfile was made. If fns=T, all functions defined as exprs (after unbreaking and unadvising) are recompiled.

brecompile asks the standard compiler questions except for OUTPUT FILE: As with bcompl, output automatically goes to file.COM, where file is the first file in files.

If coreflg=NIL, brecompile proceeds to read in each file, collecting all definitions while making the appropriate additions to nlama, nlaml, and lams, evaluating DECLARE expressions,[35] and collecting all block declarations, and other expressions which will later be copied to the output file.

---

[35] See footnote on page 18.9.

If coreflg=T, brecompile computes the block declarations from the fileVARS for each file in files.[36] Similarly, fileFNS and fileVARS are used to determine what actually appears on the files. The only access to the files is to obtain the date for each file so that it can be written onto the output file.

brecompile next processes each block declaration. If no functions in the block have been changed, the block is copied from cfile as with recompile. Otherwise, the entire block is recompiled. For pseudo-block declarations of the form (NIL fn1 ...), all variable assignments are made, but only those functions so indicated by fns are recompiled.

As with recompile, the order in which functions appear on the file must not be changed unless all of the functions that are moved also recompiled.

After completing the block declarations, brecompile processes all functions not appearing in a declaration, recompiling only those dictated by fns, and copying the compiled definitions of the remaining from cfile.

Finally, brecompile writes the portion of file.COM

---

[36] See footnote on page 18.11.

corresponding to the non-DEFINEQ expressions. If coreflg=NIL; brecompile simply writes out those expressions which it had previously collected. Otherwise, it uses fileVARS to determine what is on each file and writes the corresponding expressions on to the output file.

The value of brecompile is the output file.

If cfile= NIL, file.COM is used.[37] In addition, if fns and coreflg are both NIL, they are set to T. This is the standard usage desscribed earlier.

18.12 Compiler Structure

The compiler has two principal passes. The first compiles its input into a macro assembly language called LAP. The second pass expands the LAP code, producing (numerical) machine language instructions. The output of the second pass is written on a file and/or stored in binary program space.

Input to the compiler is usually a standard INTERLISP S-expression function definition. However, machine language coding can be included within a function by the use of one or more assemble forms. In other words, assemble allows the user to write protions of a function in LAP. Note that assemble is only a compiler directive; it has no independent definition. Therefore, functions which use assemble must be compiled in order to run.

---

[37] See footnote on page 18.11.

18.13 Assemble

The format of assemble is similar to that of PROG: (ASSEMBLE V S_1 S_2 . . . S_N). V is a list of variables to be bound during the first pass of the compilation, *not* during the running of the object code. The assemble statements S_1 ... S_N are compiled sequentially, each resulting in one or more instructions of object code. When run, the value of the assemble 'form' is the contents of AC1 at the end of the execution of the assemble instructions. Note that assemble may appear anywhere in an INTERLISP function. For example, one may write:

```
(IGREATERP (IQUOTIENT (LOC (ASSEMBLE NIL
                                     (MOVEI 1 , -5)
                                     (JSYS 13)))
                      1000)
           4)
```

to test if job runtime exceeds 4 seconds.

Assemble Statements

If an assemble statement is an atom, it is treated as a label identifying the location of the next statement that will be assembled.[38] Such labels defined in an assemble form are like prog labels in that they may be referenced from the current and lower level nested progs or assembles.

If an assemble statement is not an atom, car of the statement must be an atom and one of the following: (1) a number; (2) a LAP op-def (i.e. has a property value OPD); (3) an assembler macro (i.e. has a property value AMAC); or (4) one of the special assemble instructions given below, e.g. C, CQ, etc. Anything else will cause the error message OPCODE? - ASSEMBLE.

---

[38] A label can be the last thing in an assemble form, in which case it labels the location of the first instruction *after* the assemble form.

The types of assemble statements are described here in the order of priority used in the _assemble_ processor; that is, if an atom has both properties OPD and AMAC, the OPD will be used. Similarly a special _assemble_ instruction may be redefined via an AMAC. The following descriptions are of the first pass processing of _assemble_ statements. The second pass processing is described in the section on LAP, page 18.40.

(1) numbers - If _car_ of an assemble statement is a number, the statement is not processed in the first pass. (See page 18.40.)

(2) LAP op-defs - The property OPD is used for two different types of op-defs: PDP-10 machine instructions, and LAP macros. If the OPD definition (i.e. the property value) is a number, the op-def is a machine instruction. When a machine instruction, e.g. HRRZ, appears as _car_ of an assemble statement, the statement is not processed during the first pass but is passed to LAP. The forms and processing of machine instructions by LAP are described on page 18.41.

If the OPD definition is not a number, then the op-def is a LAP macro. When a LAP macro is encountered in an assemble statement, its arguments are evaluated and processing of the statement with evaluated arguments is left for the second pass and LAP. For example, LDV is a LAP macro, and (LDV (QUOTE X) SP) in assemble code results in (LDV X N) in the LAP code, where N is the value of SP.

The form and processing of LAP macros are described on page 18.43.

(3) assemble macros - If car of an assemble statement has a property AMAC, the statement is an assemble macro call. There are two types of assemble macros: lambda and substitution. If car of the macro definition is the atom LAMBDA, the definition will be *applied* to the arguments of the call and the resulting list of statements will be assembled. For example, repeat could be a LAMBDA macro with two arguments, n and m, which expands into n occurrences of m, e.g. (REPEAT 3 (CAR1)) expands to ((CAR1) (CAR1) (CAR1)). The definition (i.e. value of property AMAC) for repeat is:

```
(LAMBDA (N M)
  (PROG (YY)
    A   (COND
          ((ILESSP N 1)
            (RETURN (CAR YY)))
          (T (SETQ YY (TCONC YY M))
             (SETQ N (SUB1 N))
             (GO A)))))
```

If car of the macro definition is not the atom LAMBDA, it must be a list of dummy symbols. The arguments of the macro call will be substituted for corresponding appearances of the dummy symbols in cdr of the definition, and the resulting list of statements will be assembled.[39] For example, ubox could be a substitution macro which takes one argument, a number, and expands into instructions to compile the unboxed value of this number and put the result on the number stack.

The definition of UBOX is:

```
  ((E)
(CQ (VAG E))
(PUSH NP , 1))
```

---

[39] Note that assemble macros produce a list of statements to be assembled, whereas compiler macros produce a single expression. An assemble macro which *computes* a list of statements begins with LAMBDA and may be *either* spread or no-spread. The analogous compiler macro begins with an atom, (i.e. is always no-spread) and the LAMBDA is understood.

Thus (UBOX (ADD1 X)) expands to:

```
((CQ (VAG (ADD1 X)))
 (PUSH NP , 1))
```

(4) special assemble statements -

(CQ s_1 s_2 ...) CQ (compile quote) takes any number of arguments which are assumed to be regular S-expressions and are compiled in the normal way. E.g.

```
(CQ (COND ((NULL Y) (SETQ Y 1)))
    (SETQ X (IPLUS Y Z)))
```

Note: to avoid confusion, it is best to have as much of a function as possible compiled in the normal way, e.g. to load the value of x to AC1, (CQ X) is preferred to (LDV (QUOTE X) SP).

(C s_1 s_2 ...) C (compile) takes any number of arguments which are first evaluated, then compiled in the usual way. Both C and CQ permit the inclusion of regular compilation within an assemble form.

(E e_1 e_2 ...) E (evaluate) takes any number of arguments which are evaluated in sequence. For example, (PSTEP) calls a function which increments the compiler variable SP.

(SETQ var) Compiles code to set the variable var to the contents of AC1.

(FASTCALL fn) Compiles code to call fn. Fn must be one of the SUBR's that expects its arguments in the accumulators, and not on the push-down stack.

Currently, these are <u>cons</u>, and the boxing and unboxing routines.[40]

Example:

```
(CQ X)
(LDV2 (QUOTE Y) SP 2)
(FASTCALL CONS)
```

and cons[x,y] will be in AC1.

(* ...) * is used to indicate a comment; the statement is ignored.

<u>COREVALS</u>

There are several locations in the basic machine code of INTERLISP which may be referenced from compiled code. The current value of each location is stored on the property list under the property COREVAL.[41] Since these locations may change in different reassemblies of INTERLISP, they are written symbolically on compiled code files, i.e. the name of the corresponding COREVAL is written, not its value. Some of the COREVALs used frequently in <u>assemble</u> are:

CONS	entry to function CONS
LIST	entry to function LIST
KT	contains (pointer to) atom T
KNIL	contains (pointer to) atom NIL
MKN	routine to box an integer
MKFN	routine to box floating number
IUNBOX	routine to unbox an integer

---

[40] <u>list</u> may also be called with <u>fastcall</u> by placing its arguments on the pushdown stack, and the *number* of arguments in AC1.

[41] The value of <u>corevals</u> is a list of all atoms with COREVAL properties.

FUNBOX   routine to unbox floating number

The index registers used for the push-down stack pointers are also included as COREVALS. These are not expected to change, and are not stored symbolically on compiled code files; however, they should be referenced symbolically in assemble code. They are:

PP	parameter stack
CP	control stack
NP	number stack

18.14 LAP

LAP (for LISP assembly Processor) expands the output of the first pass of compilation to produce numerical machine instructions.

LAP Statements

If a LAP statement is an atom, it is treated as a label identifying the location of the next statement to be processed. If a LAP statement is not an atom, car of it must be an atom and one of the following: (1) a number; (2) a machine instruction; or (3) a LAP macro.

(1) numbers - If car of a LAP statement is a number, a location containing the number is produced in the object code.

```
e.g.     (ADD 1 , A (1))
           .
           .
           .
   A     (1)
         (4)
         (9)
```

Statements of this type are processed like machine instructions, with the initial number serving as a 36-bit op-code.

(2) Machine Instructions - If car of a LAP statement has a numeric value for the property OPD,[42] the statement is a machine instruction. The general form of a machine instruction is:

(opcode ac , @ address (index))

Opcode is any PDP-10 instruction mnemonic or INTERLISP UUO.[43]

Ac, the accumulator field, is optional. However, if present, it *must* be followed by a comma. Ac is either a number or an atom with a COREVAL property. The low order 4 bits of the number or COREVAL are OR'd to the AC field of the instruction.

@ may be used anywhere in the instruction to specify indirect addressing (bit 13 set in the instruction) e.g. (HRRZ 1 , @ ' V).

Address is the address field which may be any of the following:

= constant — Reference to an unboxed constant. A location containing the unboxed constant will be created in a region at the end of the function, and the address of the location containing the constant is

---

[42] The value is an 18 bit quantity (rather than 9), since some UUO's also use the AC field of the instruction.

[43] The TENEX JSYS's are not defined, that is, one must write (JSYS 107) instead of (KFORK).

placed in the address field of the current instruction. The constant may be a number e.g. (CAME 1 , = 3596); an atom with a property COREVAL (in which case the constant is the value of the property, at LOAD time); any other atom which is treated as a label (the constant is then the address of the labeled location) e.g. (MOVE 1 , = TABLE) is equivalent to (MOVEI 1 , TABLE); or an expression whose value is a number.

' pointer — The address is a reference to a INTERLISP pointer, e.g. a list, number, string, etc. A location containing the pointer is assembled at the end of the function, and the current instruction will have the address of this location. E.g. (HRRZ 1 , ' "IS NOT DEFINED")
(HRRZ 1 , ' (NOT FOUND))

* — Specifies the current location in the compiled function; e.g. (JRST * 2) has the same effect as (SKIPA).

literal atom — If the atom has a property COREVAL, it is a reference to a system location, e.g. (SKIPA 1 , KNIL), and the address used is the value of the coreval. Otherwise the atom is a label referencing a location in the LAP code, e.g. (JRST A).

number — The number is the address; e.g.

(MOVSI 1 , 400000Q)

(HLRZ 2 , 1 (1))

list — The form is evaluated, and its value is the address.

Anything else in the address field causes an error message, e.g. (SKIPA 1 , KNILL) - LAPERROR. A number may follow the address field and will be added to it, e.g. (JRST A 2).

Index is denoted by a *list* following the address field, i.e. the address field *must* be present if an index field is to be used. The index (car of the list) must be either a number, or an atom with a property COREVAL, e.g. (HRRZ 1 , 0 (1)) or (ANDM 1 , -1 (NP))

(3) LAP macros - If car of a LAP statement is the name of a LAP macro, i.e. has the property OPD, the statement is a macro call. The arguments of the call follow the macro name: e.g. (LQ2 FIE 3).

LAP macro calls comprise most of the output of the first pass of the compiler, and may also be used in assemble. The definitions of these macros are stored on the property list under the property OPD, and like assembler macros, may be either lambda or substitution macros. In the first case, the macro definition is applied to the arguments of the call;[44] in the second case, the arguments of the call are substituted for occurrences of the

---

[44] The arguments were already evaluated in the first pass, see page 18.36.

dummy symbols in the definition. In both cases, the resulting list of statements is again processed, with macro expansion continuing till the level of machine instructions is reached.

Some examples of LAP macros are shown in Figure 18-2.

```
(DEFLIST(QUOTE(
  (SVN ((N P)                                    (* STORE VARIABLE NAME)
        (MOVE 1 , ' N)
        (HRLM 1 , P (PP))))
  (SVB ((N)                                      (* STORE VARIABLE NAME AND VALUE)
        (HRL 1 , ' N)
        (PUSH PP , 1)))
  (LQ ((X)                                       (* LOAD QUOTE TO AC1)
       (HRRZ 1 , ' X)))
  (LQ2 ((X AC)                                   (* LOAD QUOTE TO AC)
        (HRRZ AC , ' X)))
  (LDV ((A SP)                                   (* LOAD LOCAL VARIABLE TO AC1)
        (HRRZ 1 , (VREF A SP))))
  (STV ((A SP)                                   (* SET LOCAL VARIABLE FROM AC1)
        (HRRM 1 , (VREF A SP))))
  (LDV2 ((A SP AC)                               (* LOAD LOCAL VARIABLE TO AC)
         (HRRZ AC , (VREF A SP))))
  (LDF ((A SP)                                   (* LOAD FREE VARIABLE TO AC1)
        (HRRZ 1 , (FREF A SP))))
  (STF ((A SP)                                   (* SET FREE VARIABLE FROM AC1)
        (HRRM 1 , (FREF A SP))))
  (LDF2 ((A SP)                                  (* LOAD FREE VARIABLE TO AC)
         (HRRZ 2 , (FREF A SP))))
  (CAR1 (NIL                                     (* CAR OF AC1 TO AC1)
         (HRRZ 1 , 0 (1))))
  (CDR1 (NIL                                     (* CDR OF AC1 TO AC1)
         (HLRZ 1 , 0 (1))))
  (CARQ ((V)                                     (* CAR QUOTE)
         (HRRZ 1 , @ ' V)))
  (CARQ2 ((V AC)                                 (* CAR QUOTE TO AC)
          (HRRZ AC , @ ' V)))
  (CAR2 ((AC)                                    (* CAR OF AC TO AC)
         (HRRZ AC , 0 (AC))))
  (RPQ ((V)                                      (* RPLACA QUOTE)
        (HRRM 1 , @ ' V)
  (CLL ((NAM N)                                  (* CALL FN WITH N ARGS GIVEN)
        (CCALL N , ' NAM)))
  (LCLL ((NAM N)                                 (* LINKED CALL WITH N ARGS)
         (LNCALL N , (MKLCL NAM))))
  (STE ((TY)                                     (* SKIP IF TYPE EQUAL)
        (PSTE1 TY)))
  (STN ((TY)                                     (* SKIP IF TYPE NOT EQUAL)
        (PSTN1 TY)))
  (RET (NIL                                      (* RETURN FROM FN)
        (POPJ CP ,)
  (PUSHP (NIL (PUSH PP , 1)))
  (PUSHQ ((X)                                    (* PUSH QUOTE)
          (PUSH PP , ' X)))
))(QUOTE OPD))
```

Figure 18-2

Examples of LAP Macros

18.15 Using Assemble

In order to use assemble, it is helpful to know the following things about how compiled code is run. All variable bindings and temporary values are stored on the parameter pushdown stack. When a compiled function is entered, the parameter pushdown list contains, in ascending order of address:

1. bindings of arguments to the function, where each binding occupies one word on the stack with the variable name in the left half and the value in the right half.
2. pointers to the most recent bindings of free variables used in the function.

The parameter push-down list pointer, index register PP, points to the last free variable pointer on the stack.

Temporary values, PROG and LAMBDA bindings, and the arguments to functions about to be called, are pushed on the stack following the free variable pointers. The compiler uses the value of the variable SP to keep track of the *number* of stack positions in use beyond the last free variable pointer, so that it knows where to find the arguments and free variable pointers. The function PSTEP adds 1 to SP, and PSTEPN(N) adds N to SP (N can be positive or negative).

The parameter stack should only be used for storing pointers. In addition, anything in the left half of a word on the stack is assumed to be a variable name (see Section 12). To store unboxed numbers, use the number stack, NP. Numbers may be PUSH'ed and POP'ed on the number stack.

18.16 Miscellaneous

The value of a function is always returned in AC1. Therefore, the pseudo-function, ac, is available for obtaining the current contents of AC1. For example (CQ (FOO (AC))) compiles a call to FOO with the current contents of AC1 as argument, and is equivalent to:

```
(PUSHP)
(E (PSTEP))
(CLL (QUOTE FOO) 1)
(E (PSTEPN -1))
```

In using ac, be sure that it appears as the first argument to be evaluated in the expression. For example: (CQ (IPLUS (LOC (AC)) 2))

* * *

There are several ways to reference the values of variables in assemble code. For example:

to put value of X in AC1: (CQ X)

to put value of X in AC3: (LDV2 (QUOTE X) SP 3)

to set X to contents of AC1: (SETQ X)

to set X to contents of AC2:

```
(E (STORIN (LIST (QUOTE HRRM) 2 (QUOTE ,)
                 (LIST (VARCOMP (QUOTE X))
                       (QUOTE X)
                        SP))))
```

to box and unbox a number:

(CQ (LOC (AC)))	box contents of AC1
(FASTCALL MKN)	box contents of AC1
(FASTCALL MKFN)	floating box contents of AC1
(CQ (VAG X))	unboxed value of X to AC1
(FASTCALL IUNBOX)	unbox contents of AC1
(FASTCALL FUNBOX)	floating unbox of AC1

To call a function directly, the arguments must be pushed on the parameter stack, and SP must be updated, and then the function called: e.g.

```
(CQ (CAR X))
(PUSHP)                    (* stack first argument)
(E (PSTEP))
(PUSHQ 3.14)
(E (PSTEP))                (* stack second argument)
(CLL (QUOTE FUM) 2)        (* call FUM with 2 arguments)
(E (PSTEPN -2))            (* adjust stack count)
```

and is equivalent to:

```
(CQ (FUM (CAR X) 3.14))
```

18.17 Compiler Printout and Error Messages

For each function compiled, whether from tcompl, recompile, or compile, the compiler prints:

(fn COMPILING)
(fn (arg_1 ... arg_n) ($free_1$ ... $free_n$))

The first message is printed when the compilation of fn begins. The second message is printed at the beginning of the second pass of the compilation of fn. (arg_1 ... arg_n) is the list of arguments to fn, and ($free_1$ ... $free_n$) the list of free variables referenced or set in fn.[45] The appearance of non-variables, e.g. function names, words from a comment, etc. in ($free_1$ ... $free_n$) is a good indication of parenthesis errors.

If the compilation of fn causes the generation of one or more gensym functions (see page 18.16), compiler messages will be printed for these functions between the first message and the second message for fn, e.g.

---

[45] Does not include variables on globalvars, see page 18.6.

```
(FOO COMPILING)
(FOOA0027 COMPILING)
(FOOA0027 NIL (X))
(FOO (X) NIL)
```

The compiler output for block compilation is similar to normal compilation. The pass one message, i.e. (fn compiling) is printed for each *function* in the block. Then a second pass message is printed for the entire block.[46] Then both messages are printed for each *entry* to the block.

In addition to the above output, both recompile and brecompile print the name of each function that is being copied from the old compiled file to the new compiled file. The normal compiler messages are printed for each function that is actually compiled.

Compiler Error Messages

Messages describing errors in the function being compiled are also printed on the teletype. These messages are always preceded by *****. Unless otherwise indicated below, the compilation will continue.

((form) - NON ATOMIC CAR OF FORM)

If user intended to treat the value of form as a function, he should use apply*. form is compiled as if apply* had been used. See Section 8.

(fn - NO LONGER INTERPRETED AS FUNCTIONAL ARGUMENT)

The compiler has assumed fn is the name of a function. If the user

---

[46] The names of the arguments to the block are generated by suffixing '#' and a number to the block name, e.g. (FOOBLOCK (FOOBLOCK#0 FOOBLOCK#1) free-variables).

intended to treat the *value* of fn as a function, he must use apply*. See Section 8.[47]

(tg - MULTIPLY DEFINED TAG)

tg is a PROG label that is defined more than once in a single PROG. The second definition is ignored.

(tg - UNDEFINED TAG)

tg is a PROG label that is referenced but not defined in a PROG.

(tg - MULTIPLY DEFINED TAG, ASSEMBLE)

tg is a label that is defined more than once in an assemble form.

(tg - UNDEFINED TAG, ASSEMBLE)

tg is a label that is referenced but not defined in an ASSEMBLE form.

(tg - MULTIPLY DEFINED TAG, LAP)

tg is a label that was encountered twice during the second pass of the compilation. If this error occurs with no indication of a multiply defined tag during pass one, the tag is in a LAP macro.

(tg - UNDEFINED TAG, LAP)

tg is a label that is referenced during the second pass of compilation and is not defined. LAP treats tg as though it were a coreval, and continues the compilation.

---

47 This message is printed when fn is not defined, and is also a local variable of the function being compiled. Note that earlier versions of the INTERLISP compiler did treat fn as a functional argument, and compiled code to evaluate it.

(fn - USED AS ARG TO NUMBER FN?)

The value of a predicate, such as GREATERP or EQ, is used as an argument to a function that expects numbers, such as IPLUS.

(x - IS GLOBAL)

x is on globalvars, and is also rebound in the function being compiled, either as an argument or as a local variable. The error message is to alert the user to the fact that other functions will not see this binding, since x is always accessed directly through its value cell.

(op - OPCODE? - ASSEMBLE)

op appears as car of an assemble statement, and is illegal. See page 18.35-39 for legal assemble statements.

(blkname - USED BLKAPPLY WHEN NOT APPLICABLE)

blkapply is used in the block blkname, but there are no blkapplyfns or entries declared for the block.

(fn - ILLEGAL RETURN)

return encountered when not in prog.

(tg - ILLEGAL GO)

go encountered when not in a prog.

(fn NOT COMPILEABLE)

An expr definition for fn could not be found. In this case, no code is produced for fn, and the compiler proceeds to the next function to be compiled, if any.

fn NOT COMPILEABLE.

Same as above except generates an error, thereby aborting all compilation. For example, this error condition occurs if fn is one of the functions in a block.

fn NOT FOUND.

Occurs when recompile or brecompile try to copy the compiled definition of fn from cfile, and cannot find it. See page 18.12. Generates an error.

fn NOT ON BLKFNS.

fn was specified as an entry to a block, or else was on blkapplyfns, but did not appear on the blkfns. Generates an error.

fn CAN'T BE BOTH AN ENTRY AND THE BLOCK NAME.

Generates an error.

(fn NOT IN FILE - USING DEFINITION IN CORE)

on calls to bcompl and brecompile.

Index for Section 18

Page Numbers

Page Numbers

SECTION 19[1]

ADVISING

The operation of advising gives the user a way of modifying a function without necessarily knowing how the function works or even what it does. Advising consists of modifying the *interface* between functions as opposed to modifying the function definition itself, as in editing. break, trace, and breakdown, are examples of the use of this technique: they each modify user functions by placing relevant computations ***between*** the function and the rest of the programming environment.

The principal advantage of advising, aside from its convenience, is that it allows the user to treat functions, his or someone else's, as "black boxes," and to modify them without concern for their contents or details of operations. For example, the user could modify sysout to set sysdate to the time and date of creation by advise[SYSOUT;(SETQ SYSDATE (DATE))]

As with break, advising works equally well on compiled and interpreted functions. Similarly, it is possible to effect a modification which only operates when a function is called from some other specified function, i.e., to modify the interface between two particular functions, instead of the interface between one function and the rest of the world. This latter feature is especially useful for changing the *internal* workings of a system function.

---

1 Advising was developed and implemented by W. Teitelman.

For example, suppose the user wanted time (Section 21) to print the results of his measurements to the file FOO instead of the teletype. He could accomplish this by ADVISE(((PRIN1 PRINT SPACES) IN TIME) BEFORE (SETQQ U FOO))

Note that advising prin1, print, or spaces directly would have affected all calls to these very frequently used function, whereas advising ((PRIN1 PRINT SPACES) IN TIME) affects just those calls to prin1, print, and spaces from time.

Advice can also be specified to operate after a function has been evaluated. The value of the body of the original function can be obtained from the variable !value, as with break1. For example, suppose the user wanted to perform some computation following each sysin, e.g. check whether his files were up to date. He could then:

```
ADVISE(SYSOUT AFTER (COND ((LISTP !VALUE) --)))
```
[2]

19.1 Implementation of Advising

The structure of a function after it has been modified several times by advise is given in the following diagram:

---

[2] After the sysin, the system will be as it was when the sysout was performed, hence the advice must be to sysout, not sysin. See Section 14 for complete discussion of sysout/sysin.

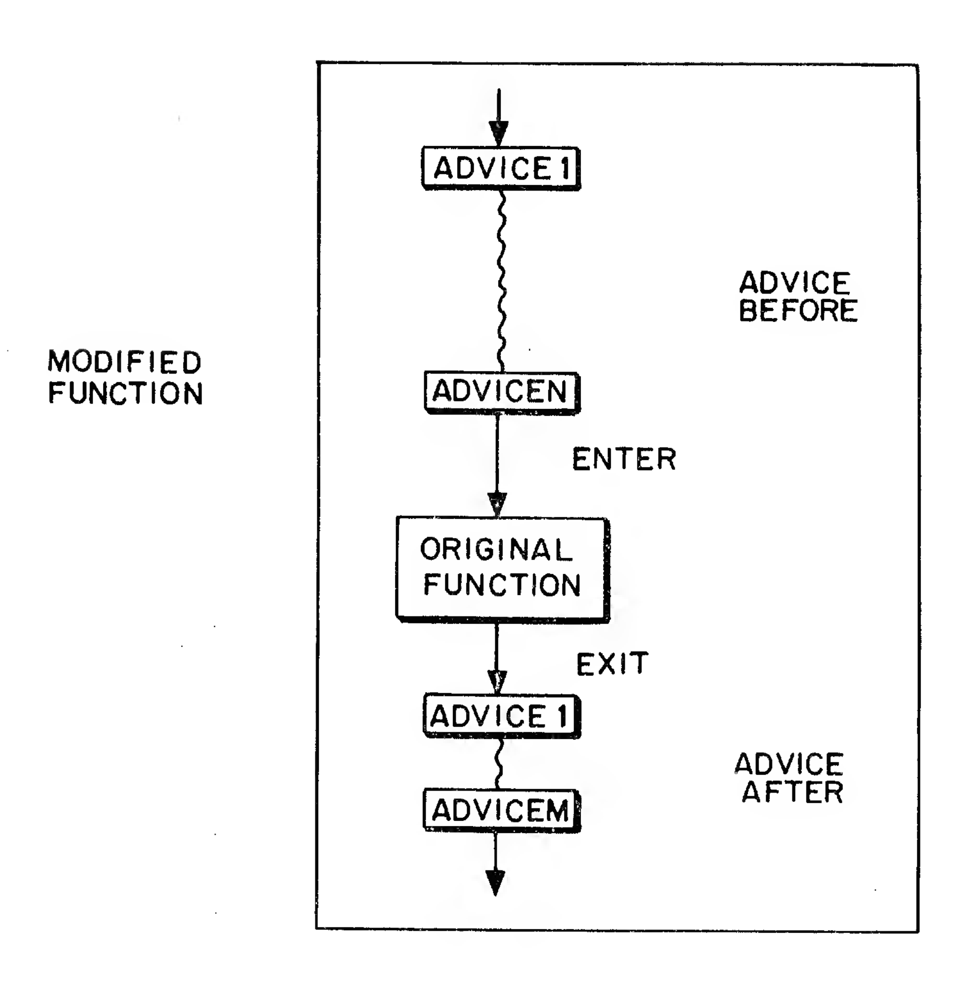

ADVICE 1
ADVICE BEFORE
MODIFIED FUNCTION
ADVICEN
ENTER
ORIGINAL FUNCTION
EXIT
ADVICE 1
ADVICE AFTER
ADVICEM

FIGURE 19-1

The corresponding INTERLISP definition is:

```
         (LAMBDA arguments (PROG (!VALUE)
              (SETQ !VALUE (PROG NIL
                   advice1
                     .
                     .                             ADVICE
                     .                             BEFORE
                   advicen
                   (RETURN form)))
              advice1
                .
                .                                  ADVICE
                .                                  AFTER
              advicem
              (RETURN !VALUE)))
```

where form is equivalent to the original definition.[3] [4]

Note that the structure of a function modified by advise allows a piece of advice to bypass the original definition by using the function RETURN. For example, if (COND ((ATOM X) (RETURN Y))) were one of the pieces of advice BEFORE a function, and this function was entered with x atomic, y would be returned as the value of the inner PROG, !value would be set to y, and control passed to the advice, if any, to be executed AFTER the function. If this same piece of advice appeared AFTER the function, y would be returned as the value of the entire advised function.

The advice (COND ((ATOM X) (SETQ !VALUE Y))) AFTER the function would have a similar effect, but the rest of the advice AFTER the function would still be executed.

---

[3] Actually, advise uses its own versions of PROG, SETQ, and RETURN, (called ADV-PROG, ADV-SETQ, and ADV-RETURN) in order to enable advising these functions.

[4] If fn was originally an EXPR, form is the body of the definition, otherwise a form using a gensym which is defined with the original definition.

19.2 Advise Functions

Advise

Advise is a function of four arguments: fn, when, where, and what. fn is the function to be modified by advising, what is the modification, or piece of advice. when is either BEFORE or AFTER, and indicates whether the advice is to operate BEFORE or AFTER the body of the function definition is evaluated. where specifies exactly where in the list of advice the new advice is to be placed, e.g., FIRST, or (BEFORE PRINT) meaning before the advice containing print, or (AFTER 3) meaning after the third piece of advice, or even (: TTY:). If where is specified, advise first checks to see if it is one of LAST, BOTTOM, END, FIRST, or TOP, and operates accordingly. Otherwise, it constructs an appropriate edit command and calls the editor to insert the advice at the corresponding location.

Both when and where are optional arguments, in the sense that they can be omitted in the call to advise. In other words, advise can be thought of as a function of two arguments [fn;what], or a function of three arguments: [fn;when;what], or a function of four arguments: [fn;when;where;what]. Note that the advice is always the *last* argument. If when=NIL, BEFORE is used. If where=NIL, LAST is used.

advise[fn;when;where;what] fn is the function to be advised, when=BEFORE or AFTER, where specifies where in the advice list the advice is to be inserted, and what is the piece of advice.

If fn is of the form (fn1 IN fn2), fn1 is changed to fn1-IN-fn2 throughout fn2, as with break, and

then fn1-IN-fn2 is used in place of fn.[5]

If fn is broken, it is unbroken before advising.

If fn is not defined, an error is generated, NOT A FUNCTION.

If fn is being advised for the first time, i.e. if getp[name,ADVISED]=NIL, a gensym is generated and stored on the property list of fn under the property ADVISED, and the gensym is defined with the original definition of fn. An appropriate S-expression definition is then created for fn.[6] Finally, fn is added to the (front of) advisedfns.[7]

If fn has been advised before, it is moved to the front of advisedfns.

The advice is inserted in fn's definition either BEFORE or AFTER the original body function

---

[5] If fn1 and/or fn2 are lists, they are distributed as shown in the example on page 19.2.

[6] Using private versions of PROG, SETQ, and RETURN, so that these functions can also be advised.

[7] So that unadvise[T] always unadvises the last function advised. See page 19.8.

depending on _when_.[8] Within that context, its position is determined by _where_. If _where_=LAST, BOTTOM, END, or NIL, the advice is added following all other advice, if any. If _where_=FIRST or TOP, the advice is inserted as the first piece of advice. Otherwise, _where_ is treated as a command for the editor, a la _breakin_, e.g. (BEFORE 3), (AFTER PRINT) .

Finally list[when;where;what] is added (by _addprop_) to the value of property ADVICE on the property list _fn_.[9] Note that this property value is a list of the advice in order of calls to _advise_, not necessarily in order of appearance of the advice in the definition of _fn_.

The value of _advise_ is _fn_.

If _fn_ is non-atomic, every function in _fn_ is advised with the same values (but copies) for _when_, _where_, and _what_. In this case, the value of _advise_ is a list of individual functions.

Note: advised functions can be broken. (However if a function is broken at the time it is advised, it is first unbroken.) Similarly, advised functions can

---

8 A special case is _when_=BIND. Here the advice is treated as a list of PROG variables to be bound. The variables are _nconced_ to the PROG variable list containing _!value_. See page 19.4.

9 So that a record of all the changes is available for subsequent use in readvising, see page 19.8.

be edited, including their advice. unadvise will still restore the function to its unadvised state, but any changes to the body of the definition will survive. Since the advice stored on the property list is the same structure as the advice inserted in the function, editing of advice can be performed on either the function's definition or its property list.

unadvise[x] — is a no-spread NLAMBDA a la unbreak. It takes an indefinite number of functions and restores them to their original unadvised state, including removing the properties added by advise.[10] unadvise saves on the list advinfolst enough information to allow restoring a function to its advised state using readvise. advinfolst and readvise thus correspond to brkinfolst and rebreak.

unadvise[] unadvises all functions on advisedfns.[11] It first sets advinfolst to NIL.

unadvise[T] unadvises the first function of advisedfns, i.e., the most recently advised function.

readvise[x] — is a no-spread NLAMBDA a la rebreak for restoring a function to its advised state without having to specify all the advise information. For each

---

[10] Except if a function also contains the property READVICE (see readvise below), unadvise moves the current value of the property ADVICE to READVICE.

[11] In reverse order, so that the most recently advised function is unadvised last.

function on x, readvise retrieves the advise information either from the property READVICE for that function, or from advinfolst, and performs the corresponding advise operation(s). In addition it stores this information on the property READVICE if not already there. If no information is found for a particular function, value is (fn - NO ADVICE SAVED).

readvise[] readvises everything on advinfolst.

readvise[T] readvises just the first function on advinfolst, i.e., the function most recently unadvised.

The difference between advise, unadvise, and readvise versus break, unbreak, and rebreak, is that if a function is not rebroken between successive unbreak[]'s, its break information is forgotten. However, once readvised, a function's advice is permanently saved on its property list (under READVICE); subsequent calls to unadvise will not remove it. In fact, calls to unadvise update the property READVICE with the current value of the property ADVICE, so that the sequence readvise, advise, unadvise causes the augmented advice to become permanent. Note that the sequence readvise, advise, readvise removes the 'intermediate advice' by restoring the function to its earlier state.

advisedump[x;flg] — Used by prettydef when given a command of the form (ADVISE --) or (ADVICE --). flg=T corresponds to (ADVISE --), i.e. advisedump writes both a deflist and a readvise. flg=NIL corresponds to (ADVICE --), i.e. only the deflist is written. In either case, advisedump copies the advise information to

the property READVICE, thereby making it 'permanent' as described above.

Index for Section 19

SECTION 20

PRINTSTRUCTURE AND INTERSCOPE

20.1 Printstructure[1]

In trying to work with large programs, a user can lose track of the hierarchy which defines his program structure; it is often convenient to have a map to show which functions are called by each of the functions in a system. If fn is the name of the top level function called in your system, then typing in printstructure[fn] will cause a tree printout of the function-call structure of fn. To illustrate this in more detail, we use the printstructure program itself as an example.

---

[1] A preliminary version of printstructure was written by D. G. Bobrow. The current form of printstructure was written by W. Teitelman.

```
PRINTSTRUCTURE      PRGETD
                    PROGSTRUC PRGETD
                              PRGSTRC   NOTFN     PRGETD
                                        PROGSTRUC
                                        PRGSTRC1  PRNCONC
                                                  PRGSTRC1
                                                  PRGSTRC
                                        PRNCONC
                                        PRGSTRC
                              CALLS1    MAKELIST
                                        NOTFN
                                        CALLS2    CALLS1
                                                  PRGETD
                    TREEPRINT TREEPRINT1
                              TREEPRINT
                    VARPRINT  VARPRINT1 TREEPRINT1
                              VARPRINT2 ALLCALLS  ALLCALLS1 ALLCALLS1
                              TREEPRINT1

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

PRINTSTRUCTURE      [X,FILE: DONELST,N,TREELST,TREEFNS,LSTEM,X,Y,Z,
FN,TREE,PRDEPTH,LAST-PRINTSTRUCTURE]
   CALLED BY:

PRGETD   [X,FLG; ; ]
   CALLED BY: PRINTSTRUCTURE,PROGSTRUC,NOTFN,CALLS2

PROGSTRUC [FN,DEF; N,Y,Z,CALLSFLG,VARSFLG,VARS1,VARS2,D,X; N,DONELST]
   CALLED BY: PRINSTRUCTURE,PRGSTRC

PRGSTRC  [X,HEAD,FLG; Y,TEM,X; VARSFLG,D,NOFNS,CALLSFLG,N,DONELST,
TREEFNS,NOTRACEFNS,FN,VARS1,QUOTEFNS]
   CALLED BY: PROGSTRUC,PRGSTRC1,PRGSTRC

NOTFN    [FN; DEF; NOFNS,YESFNS,FIRSTLOC,LASTLOC]
   CALLED BY: PRGSTRC,CALLS1

PRGSTRC1 [L,HEAD,FLG; A,B; VARS1,VARS2]
   CALLED BY: PRGSTRC,PRGSTRC1

PRNCONC  [X,Y; ; CALLSFLG]
   CALLED BY: PRGSTRC1,PRGSTRC

CALLS1   [ADR,GENFLG,D; LIT,END,V1,V2,LEFT,OPD,X,X; VARS1,VARS2,
VARSFLG]
   CALLED BY: PROGSTRUC,CALLS2

MAKELIST [N,ADR; L; ]
   CALLED BY: CALLS1
```

Figure 20-1

The upper portion of this printout is the usual horizontal version of a tree. This tree is straighforwardly derived from the definitions of the functions: printstructure calls prgetd, progstruc, treeprint, and varprint. progstruc in turn calls prgetd, prgstrc and calls1. prgstrc calls notfn, progstruc, prgstrc1, prnconc, and itself. prgstrc1 calls prnconc, itself, and prgstrc. Note that a function whose substructure has already been shown is not expanded in its second occurrence in the tree.

The lower portion of the printout contains, for each function, information about the variables it uses, and a list of the functions that call it. For example, printstructure is a function of two arguments, x and file. It binds eleven variables internally: donelst, n, ... tree,[2] and uses prdepth and last-printstructure as free variables. It is not called by any of the functions in the tree. prgetd is a function of two arguments, x and flg, binds no variables internally, uses no free variables, and is called by printstructure, progstruc, notfn and calls2.

printstructure calls many other low-level functions such as getd, car, list, nconc, etc. in addition to the four functions appearing in the above output. The reason these do not appear in the output is that they were defined "uninteresting" by the user for the purposes of his analysis. Two functions, firstfn and lastfn, and two variables, yesfns and nofns are used for this purpose. Any function that appears on the list nofns is not of interest, any function appearing on yesfns is of interest.

yesfns=T effectively puts all functions on yesfns. As for functions appearing on neither nofns or yesfns, all interpreted functions are deemed interesting, but only those compiled functions whose code lies in that portion of bpspace

---

[2] Variables are bound internally by either PROGs LAMBDA-expressions.

between the two limits established by firstfn and lastfn. For example, the above analysis was performed following firstfn[PRINTSTRUCTURE] and lastfn[ALLCALLS1].

Three other variables, notracefns, quotefns, and prdepth also affect the action of printstructure. Functions that appear on the list notracefns will appear in the tree, assuming they are "interesting" functions as defined above, but their definitions will not be analyzed.

Functions that appear on quotefns are analyzed, assuming they are "interesting," but when they appear as car of a form, the rest of the form, i.e., the arguments, is not analyzed. For example, if the function prinq were defined as (NLAMBDA (X) (MAPC X (FUNCTION PRIN1))) and included on quotefns, and the form (PRINQ (NOW IS THE TIME)) appeared in a function being analyzed, prinq *would* appear in the tree and be analyzed but the 'form' (NOW IS THE TIME) would be skipped. The initial setting of quotefns is NLAMBDAs, which effectively includes all NLAMBDAs (functions with argtype 1 or 3) on quotefns, except for those functions which printstructure knows require evaluation, e.g., ersetq, nlsetq, or, and, etc. The arguments to these functions are always analyzed.

Finally, prdepth is a cutoff depth for analysis. It is initially set to 7.

printstructure has incorporated in it the necessary information for analyzing non-standard forms such as cond, prog and selectq. It is also capable of

analyzing compiled or interpreted functions equally well.[3] [4] In the case of compiled functions, printstructure will automatically analyze any functions generated by the compiler, such as those caused by compiling forms beginning with ersetq, nlsetq, or function.

If printstructure encounters a form beginning with two left parentheses in the course of analyzing an interpreted function (other than a COND clause or open lambda expression) it notes the presence of a possible parentheses error by the abbreviation P.P.E., followed by the function in which the form appears, and the form itself, as in the example below. Note also that since printstructure detects functions that are not defined, (i.e., atoms appearing as CAR of a form), printstructure is a useful tool for debugging.

---

[3] except there may be some confusion in analyzing compiled functions, if the name of a variable and a function are the same. For this reason, it is best to printstructure the interpreted version of a function whenever possible.

[4] Printstructure knows about CLISP (Section 23) to the extent that if it encounters untranslated iterative statements or IF-THEN-ELSE statements, it will automatically dwimify them before analyzing them.

```
←PP FOO

(FOO
  [LAMBDA (X)
    (COND
      ((CAR X) (FOO1 X))
      (T ((CONS X (CAR X])
FOO
←PRINTSTRUCTURE(FOO)

FOO        FOO1

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

FOO        [X; ; ]
   CALLED BY:

FOO1       IS NOT DEFINED.

P.P.E. IN FOO - ((CONS X (CAR X)))
```

Figure 20-2

Other Options

printstructure is a function of three arguments, x, exprflg, and file. printstructure analyzes x, sets the free variable last-printstructure to the results of its analysis, prints the result (in the format shown earlier) to file (which is opened if necessary and closed afterwards), and returns x as its value. Thus if the user did not want to see any output, he could call printstructure with file=NIL:,[5] and then process the result himself by using last-printstructure.

printstructure always checks for EXPR properties on the property list of functions that are not defined. However, if exprflg=T, printstructure will prefer to analyze EXPR definitions whenever possible, i.e. if the function

---

[5] NIL: is a TENEX output device that acts like a 'bottomless pit'. Note that file=NIL (not NIL:) means print the tree to primary output file.

definition call contains a compiled definition, and there is also an EXPR property, the latter will be analyzed. See footnote on page 20.5.

x can be NIL, a list, a function, or an atom that evaluates to a list. If x is NIL, printstructure does not perform any analysis, but simply prints the result of the last analysis, i.e., that stored on last-printstructure. Thus the user can effectively redirect the output that is going to the terminal to a disc file by aborting the printout, and then performing printstructure[NIL;file].

If x is a list, printstructure analyzes the first function on x, and then analyzes the second function, *unless* it was already analyzed, then the third, etc., producing however many trees required. Thus, if the user wishes to analyze a *collection* of functions, e.g., breakfns, he can simply perform (PRINTSTRUCTURE BREAKFNS).

If x is not a list, but is the name of a function, printstructure[x] is the same as printstructure[(x)]. Finally, if the value of x is a list of functions, printstructure will process that list as described above.

Note that in the case that x is a list, or evaluates to a list, subsequent functions are *not* separately analyzed if they have been encountered in the analysis of a function appearing earlier on the list. Thus, the ordering of x can be important. For example, if both FOO and FIE call FUM, printstructure[(FOO FIE FUM)], will produce a tree for FOO containing embedded in it the tree for FUM. FUM will not be expanded in the tree for FIE, nor will it have a tree of its own. (Of course, if FOO also calls FIE, then FIE will not have a tree either.) The convention of listing FUM can be used to *force* printstructure to give FUM a tree of its own. Thus printstructure[(FOO FIE (FUM))] will produce *three* trees, and neither of the calls to FUM from FOO or FIE will be expanded in their respective trees. Of course, in this example, the same effect could have been achieved by

reordering, i.e., printstructure[(FUM FOO FIE)]. However, if FOO, FIE, and FUM, all called each other, and yet the user wanted to see three separate trees, no ordering would suffice. Instead, the user would have to do printstructure[((FOO) (FIE) (FUM))].

The result of the analysis of printstructure is in two parts: donelst, a list summarizing the argument/variable information for each function appearing in the tree(s), and treelst, a list of the trees. last-printstructure is set to cons[donelst;treelst].

donelst is a list consisting, in alternation, of the functions appearing in any tree, and a variable list for that function. car of the variable list is a list of variables bound in the function, and cdr is a list of those variables used freely in the function. Thus the form of donelst for the earlier example would be:

```
(PRINTSTRUCTURE ((X FILE DONELST N TREELST TREEFNS L TEM X Y Z
FN TREE) PRDEPTH LAST-PRINTSTRUCTURE) PRGETD ((X FLG))
PROGSTRUC (( FN DEF N Y Z CALLSFLG VARSFLG VARS1 VARS2 D X)
N DONELST) ... ALLCALLS1 ((FN TR A B)))
```

Possible parentheses errors are indicated on donelst by a non-atomic form appearing where a function would normally occur, i.e., in an odd position. The non-atomic form is followed by the name of the function in which the P.P.E. occurred.

Printstructure Functions

printstructure[x;exprflg;file] — analyzes x, saves result on last-printstructure, outputs trees and variable information to file, and returns x as its value.

If exprflg=T, printstructure will prefer to analyze expr's. See page 20.6.

treeprint[x;n] — prints a tree in the horizontal fashion shown in the examples above. i.e., printstructure performs (MAPC TREELST (FUNCTION TREEPRINT)).

varprint[donelst;treelst] — prints the "lower half" of the printstructure output.

allcalls[fn;treelst] — uses treelst to produce a list of the functions that call fn.

firstfn[fn] — If fn=T, lower boundary is set to 0, i.e., all subrs and all compiled functions will pass this test. If fn=NIL, lower boundary set at end of bpspace, i.e., no compiled functions will pass this test. Otherwise fn is the name of a compiled function and the boundary is set at fn, i.e., all compiled functions defined earlier than fn are rejected.

lastfn[fn] — if fn=NIL, upper boundary set at end of bpspace, i.e., all compiled functions will pass this test. Otherwise boundary set at fn, i.e., all compiled functions defined later than fn are rejected.

Thus to accept all compiled functions, perform firstfn[T] and lastfn[NIL]: to reject all compiled functions, perform firstfn[].

calls[fn;exprflg;varsflg] — is a fast 'one-level' printstructure, i.e., it indicates what functions fn calls, but does not go further and analyze any of them. calls does not print a tree, but reports its findings by returning as its value a list of three elements: a list of all functions called by fn, a list of variables bound in fn, and a list of variables used freely in fn, e.g.,

```
calls[progstruc] = ((PRGETD EXPRP PRGSTRC CCODEP
CALLS1 ATTACH) (FN DEF N Y Z CALLSFLG VARSFLG
VARS1 VARS2 D X) (N DONELST))
```

fn can be a function name, a definition, or a form. Calls first does firstfn(T), lastfn() so that all subrs and compiled functions appear, except those on nofns. If varsflg is T, calls ignores functions and only looks at the variables (and therefore runs much faster).

vars[fn;exprflg] — cdr[calls[fn;exprflg;T]]

freevars[fn;exprflg] — cadr[vars[fn;exprflg]]

20.2 Interscope[6]

While printstructure is a convenient tool for giving the user an *overview* of the structure of his programs, it is not well suited for determining the answer to particular questions the user may have about his programs. For example, if FOO uses X freely, and the user wants to know where X is bound 'above' FOO, he has to visually trace back up the tree that is output by printstructure, and, at each point, look down at the lower portion of the printout and find whether the corresponding function binds X. For large systems, such a procedure can be quite tedious. Furthermore, printstructure does not even compute certain certain important types of information. For example, printstructure does not distinguish between functions that *use* a variable freely and those that *set* it (or smash it).

Interscope is an extension of printstructure designed to resolve these shortcomings. Like printstructure, interscope analyses programs (functions), although it extracts considerably more information and relationships than does printstructure. However, instead of presenting the information it obtains in a predetermined format, interscope allows the user to ask it questions about the programs it has analysed, i.e. to interrogate its data base. These questions can be input in English, and contain conjunctions, disjunctions, and negations of the many relationships between functions and variables that interscope knows about. The questions can be closed questions, e.g. "DOES FOO CALL FIE?", or open questions, "WHAT FUNCTIONS CALL FIE?". The answers to some questions are obtainable directly from the data base, e.g. "WHAT VARIABLES DOES FOO SET?" Other questions cause interscope to search its data base, e.g. "WHAT FUNCTIONS BIND VARIABLES THAT FOO SETS?". Figure 20-3 contains a sample session with interscope.

---

[6] Interscope was written by P. C. Jackson.

```
←INTERSCOPE]
 Hello, shall I analyze a system?                                    [1]
&←WTFIXFNS AND CLISPFNS.
 This may take a few minutes.

GC: 8
1233, 10431 FREE WORDS
 Shall I analyze another system?
&←NO                                                                 [2]
 Ok, what would you like to know?
&WHO CALLS RETDWIM?
(WTFIX FIX89TYPEIN FIXAPPLY FIXATOM FIXCONTINUE CLISPATOM FIXT)
&HOW IS CLISPATOM CALLED?
 I didn't understand that.                                           [3]
&WHAT FUNCTIONS CALL CLISPATOM?                                      [4]
(WTFIX FIXAPPLY FIXATOM)
&WHAT FREE VARIABLES DOES CLISPATOM USE?
(ONLYSPELLFLG CLISPCHANGES CLISPFLG TYPE-IN? CLISPSTATS INFIXSTATS LST
FAULTXX CHCONLST FAULTX DWIMIFYFLG 89CHANGE FAULTPOS)
&WHO BINDS TAIL?
(WTFIX RETDWIM1 RETDWIM2 RETDWIM3 CLISPFUNCTION? CLISPATOM0 CLISPATOM1
CLISPATOM1A CLISPATOM2A DWIMIFY1A DWIMIFY2 DWIMIFY2A CLISPRESPELL)
&WHO BINDS TAIL AND CALLS CLISPATOM SOMEHOW?
(WTFIX DWIMIFY2)
&WHAT VARS DOES HELPFIX CHANGE?
(FORM LASTPOS NOCHANGEFLG HELPFIXTAIL FN TEM BRKEXP)
&WHAT FUNCTIONS CHANGE THE VARIABLE TENTATIVE?
(CLISPATOM1 CLISPATOM2 CLISPATOM2C CLISPATOM2A CLISPATOM1A)
&WHO CHANGES TAIL?
(FIXATOM HELPFIX1 CLISPATOM1 CLISPATOM2 DWIMIFY2)
&WHAT FNS USE TEM AS AN INTERANL VAR AND
...ARE CALLED BY CLISPATOM INDIRECTLY?
INTERANL=INTERNAL ?  Yes
(RETDWIM RETDWIM1 FIX89TYPEIN)
&HOW DOES CLIAPTOM CALL LISTP?
CLIAPTOM=CLISPATOM ?  Yes
((CLISPATOM LISTP) (CLISPATOM *** RETDWIM *** LISTP) (CLISPATOM      [5]
FIX89 FIX89A LISTP))
&SHOW ME THE PATHS FROM CLISPATOM TO LISTP.
CLISPATOM LISTP
          RETDWIM LISTP                                              [6]
                  RETDWIM1 LISTP
          FIX89TYPEIN RETDWIM ...
          FIX89 FIX89A LISTP
&DOES GETVARS SMASH ANY VARIABLES?
(L)
&SHOW ME HOW GETVARS SMASHES L.
 (NCONC L (AND (LISTP X) (MAPCAR & &)))
&GOODBYE.
 Goodbye.
```

Figure 20-3

In order to answer questions about programs, interscope must analyze them and build its data-base. When interscope is first called, it will ask the user what functions he wants analyzed. The user can respond to this question by giving interscope either: 1) the name of the top level function called in his system, or 2) the name of a variable that evaluates to a list of top level functions, or 3) the list itself. All of the functions below each top level function will be analyzed, except those which are declared to be "uninteresting," as described below. Note that after interscope goes into question-answering mode,[7] the user can instruct interscope to analyze additional functions, either in English input, e.g. "ANALYZE FOOFNS." or by calling the function lookat directly (page 20.17).

The structure of interscope may be divided into three major subsystems: a top-level monitor function, an English preprocessor, and the functions which build and search the data base. The monitor function is implemented via userexec (see Section 22), so that the features of the programmer's assistant are available from within interscope.[8] For example, the user can REDO or FIX interscope questions, interrogate the history list for his session, or run

---

[7] When interscope is first called, and it has not previously analyzed any functions, it is in analysis mode, as indicated by its greeting and prompt character (&← instead of &) (see [1] in Figure 20-3). Interscope goes into question-answering mode when the user answers NO to the question "Shall I analyse a (another) system?" ([2] in Figure 20-3). The only difference between analysis mode and question-answering mode is that in analysis mode, interscope treats forms as indicating a list of functions to be analysed, whereas in question-answering mode, interscope simply passes forms back to lispx for evaluation.

[8] interscope assumes that any input line terminated by a punctuation mark is intended for it to process. interscope will also attempt to process other input lines, i.e. those not ending in punctuation. However, if it is not able to make sense of the input, interscope will assume that it was intended to be handled by lispx, and pass it back for evaluation. For example, if the user types "HAS THOU SLAIN THE JABBERWOCK?", interscope will respond "I didn't understand that", but if the user omits the '?', the line will be given to lispx for evaluation and (probably) cause a U.D.F. HAS error.

programs from within interscope.[9]

The English preprocessor translates English questions,[10] statements, and commands into INTERLISP forms appropriate for searching and building the interscope data base. Although this preprocessor is fairly flexible and robust (e.g. includes spelling correction), it translates only a limited subset of English sentences, and replies "I didn't understand that." to anything outside this subset ([3] in Figure 20-3).[11] When this happens, usually a simple rephrasing of the question will suffice to allow interscope to handle it ([4] in Figure 20-3).

The interscope data-base can be accessed directly by the user via the functions described below. It should be noted that interscope actually creates two data bases, the first containing information about the elementary relations between the functions and variables in the user's system, and the second containing information derived from the first, i.e. the paths by which one function calls another. The first data base is created when interscope analyzes a system (via the function lookat). The second data base is developed incrementally (by the function paths), depending on the questions asked by the user. Both data bases are stored on the property lists of the functions and variables which are analyzed.

---

9 Since the data base that interscope constructs is global (stored on property lists), the user can also exit from interscope, either by typing OK or GOODBYE, or via control-D, and then reenter interscope at some later point and continue asking questions, without having to reanalyze his functions.

10 The translation of the most recent input is always stored in the function definition cell of the atom MEANING.

11 Where possible, interscope will try to inform the user what part of the sentence it did not understand.

Interscope "understands" a wide variety of the elementary relations that exist between functions and variables, e.g. which functions bind, use, change, test, or smash a given variable, which functions may cause a given function to be called, either directly or indirectly,[12] which variables are used as global or local free variables, either by a given function or by a group of functions, etc.

Information about the function-call paths from one program to another is "generalized" when it is stored; e.g. at [5] in Figure 20-3, one of the paths by which CLISPATOM calls LISTP is given as (CLISPATOM *** RETDWIM *** LISTP), which means that there is more than one path from CLISPATOM to RETDWIM, and more than one path from RETDWIM to LISTP.

The conventions used by interscope for recognizing functions that are "uninteresting" are the same as those used by printstructure (page 20.3), i.e. yesfns, nofns firstfn, and lastfn all have the same effect as for printstructure.

Interscope Functions

paths[x;y;type;must;avoid;only] Value is a list of paths from x to y, where each path is an ordered list of functions. *** is used to indicate multiple paths. For example, if FOO calls FIE, and FIE calls FUM directly as well as calling FIE1 which calls FUM, then paths[FOO;FUM] returns ((FOO FIE *** FUM)).

---

[12] e.g. if FOO calls FIE, and FIE calls FUM, then FOO calls FUM indirectly. 'SOMEHOW' means directly or indirectly, e.g. "WHAT FUNCTIONS CALL FOO SOMEHOW?"

type, must, avoid, and only are optional. type can be either CALLS or CALLEDBY (NIL is equivalent to CALLS), e.g. in the above example, paths[FUM;FOO;CALLEDBY] would return the same set of paths as paths[FOO;FUM], except each path would be in the reverse order.

must, avoid, and only are used to select out certain types of paths. Each can be specified by an atom which evaluates to a list of functions, or a form which evaluates to such a list. If (the value of) must is non-NIL, each path is required to go through at least one of the members of must. If avoid is non-NIL, no path can go through any member of avoid. If only is non-NIL, no path can go through any function which is not a member of only, i.e. each path can only go through functions on only.[13]

treepaths[x;y;type;must;avoid;only] Like paths, except prints paths as a tree structure, as shown at [6] in Figure 20-3. type, must, avoid, and only have the same meaning as with paths.[14]

---

[13] paths is called for English inputs of the form: "WHAT ARE THE PATHS FROM x TO y?". Such questions can be modified with subordinate clauses to indicate values for must, avoid, and/or only, e.g. "WHAT ARE THE PATHS FROM FOO TO FIE WHICH ONLY GO THROUGH FOOFNS AND AVOID FIEFNS?"

[14] treepaths is called for English inputs of the form "SHOW ME HOW x CALLS y", "DISPLAY THE PATHS FROM x TO y", etc.

lookat[x] — Builds the initial data base describing the system x, where x is either the name of a function, the name of a variable which evaluates to a list of functions, or the list of functions itself.

clumpget[object;relation;universe] — Value is a list of objects (functions or variables) which have the indicated relation with respect to object, e.g. clumpget[FOO;CALLERS] returns a list of functions that call FOO, clumpget[X;SMASHERS] a list of functions that smash the variable X, etc. A complete list of the possible values for relation is given below.[15]

object can be a list of objects (or a variable which evaluates to a list of objects), in which case the value returned by clumpget is the list of all objects which have the indicated relation to *any* of the members of object.

Similarly, universe can be a list of objects (or a variable which evaluates to a list of objects), in which case the value returned by clumpget is the list of all objects in universe which have the indicated relation to object (or any of the members of object), e.g. clumpget[X;SMASHERS;FOOFNS].

---

[15] If clumpget is given a value for relation that it does not recognize, it will attempt spelling correction, and if that fails, generate an error. If given a value for object that it has not seen before, it will type "I don't know anything about object, shall I analyse a system?" and go into analysis mode.

Finally,, universe can be a *relation*, which is equivalent to supplying clumpget[object;universe] in place of object, i.e. the value returned is the list of all objects which have the indicated relation to any of the members of the {set of all objects which bear the relationship universe to object}. For example, clumpget[FOO;CALLERS;CALLEDFNS] is a list of all functions that call any of the functions (CALLERS) that are directly called by FOO (CALLEDFNS). clumpget[FOO;FREEUSERS;LOCALVARS] is a list of functions that use freely any of the variables that are bound locally by FOO.

Currently, the following relations are implemented:

CALLERS — list of functions that directly call object.

CALLEDFNS — list of functions directly called by object.

CALLCAUSERS — list of functions that call object, perhaps indirectly. In English: "WHO CALLS FOO SOMEHOW?".

CALLSCAUSED — list of functions called by object, perhaps indirectly. In English: "WHO DOES FOO CALL SOMEHOW?"

ABOVE — union of object with CALLCAUSERS.

BELOW — union of object with CALLSCAUSED.

ARGS — arguments of object.

ARGBINDERS — list of functions that have object as an argument.

LOCALVARS — list of variables that are locally bound in object, e.g. PROG vars.

LOCALBINDERS — list of functions that bind object as a local variable.

FREEVARS — list of variables used freely by object.

FREEUSERS — list of functions that use object freely.

LOCALFREEVARS — list of variables that are used freely in object, but are bound in object before they are used, e.g. clumpget[FOO;LOCALFREEVARS;BELOW] gives a list of those variables used freely below FOO, but are bound above the place that they are used.[16] In English: "WHAT ARE THE LOCAL FREE VARS (VARIABLES) BELOW FOO?"

GLOBALFREEVARS — list of variables used freely in object without previously being bound in object.

ENTRYFNS — list of each function in object which is not called by any function in object other than itself, e.g. clumpget[FOOFNS;ENTRYFNS].

---

[16] Note that if object is the name of a function and universe is NIL, LOCALFREEVARS will always be NIL, and GLOBALFREEVARS the same as FREEVARS. It is only in connection with collections of functions that LOCALFREEVARS and GLOBALFREEVARS become interesting.

SELFRECURSIVE — list of functions in <u>object</u> which call themselves directly.

CAUSESELFCALL — list of functions in <u>object</u> which could call themselves, perhaps indirectly.

CAUSERECURSION — list of functions in <u>object</u> which cause some function to call itself, perhaps indirectly.

CHANGEVARS — list of variables that are changed by <u>object</u>, where 'changed' means *any* flavor of assignment, i.e. via SETQ, SETQQ, RPAQ, SETN, or even an expression of the form (RPLACA (QUOTE atom) value) (or FRPLACA, /RPLACA, SAVESET, etc.)[17]

CHANGERS — list of functions that change <u>object</u>.

Note: 'set' in English input means any flavor of assignment, and translates the same as 'change'.

SMASHVARS — list of variables whose value are smashed by <u>object</u>, where 'smash' means the variable appears as the first argument to one of the list of functions on <u>smasherslst</u>.[18]

---

[17] <u>clumpget</u> will accept as relations SETQVARS, SETQERS, SETVARS, SETTERS, SETQQERS, SETQQVARS, etc., in case the user wants to distinguish between the various flavors of assignments. In English, "WHAT ARE THE SETQERS OF X?", etc.

[18] Initially (RPLACA RPLACD FRPLACA FRPLACD /RPLACA /RPLACD NCONC NCONC1 /NCONC /NCONC1 ATTACH /ATTACH RPLNODE /RPLNODE RPLNODE2 /RPLNODE2). As with assignments, <u>clumpget</u> will accept as relations RPLACAERS, RPLACAVARS, RPLACDERS, RPLACDVARS, etc., in case the user wants to distinguish the different types of smashing.

SMASHERS — list of functions that smash object.

TESTVARS — list of variables that are tested by object, where 'tested' means they appear as the first argument to one of the list of functions on testerslst, initially (ATOM LISTP NUMBERP NLISTP STRINGP EQ EQP EQUAL NULL), or anywhere in an AND or OR, or as the predicate in a COND clause, or as the first argument to SELECTQ, etc.

TESTERS — list of functions that test object.

USEVARS — list of variables that are used in object, where 'used' means actually appear in the body of the function, i.e. if a variable is simply bound, but not actually used anywhere, it will not be included in the value of USEVARS. CHANGEVARS and TESTVARS are subsets of USEVARS.

USERS — list of functions that use object.

Index for Section 20

SECTION 21

MISCELLANEOUS

21.1 Measuring Functions

time[timex;timen;timetyp] is an nlambda function. It executes the computation timex, and prints out the number of conses and computation time. Garbage collection time is subtracted out.

```
←TIME((LOAD (QUOTE PRETTY) (QUOTE PROP]
FILE CREATED 7-MAY-71 12:47:14

GC: 8
582, 10291 FREE WORDS
PRETTYFNS
PRETTYVARS
3727 CONSES
10.655 SECONDS
PRETTY
```

If timen is greater than 1 (timen=NIL equivalent to timen=1), time executes timex timen number of times and prints out number of conses/timen, and computation time/timen. This is useful for more accurate measurement on small computations,e.g.

```
←TIME((COPY (QUOTE (A B C))) 10)
30/10 = 3 CONSES
.055/10 = .0055 SECONDS
(A B C)
```

If timetype is 0, time measures and prints total *real* time as well as computation time, e.g.

```
←TIME((LOAD (QUOTE PRETTY) (QUOTE PROP)) 1 0]
FILE CREATED 7-MAY-71 12:47:14

GC: 8
582, 10291 FREE WORDS
PRETTYFNS
PRETTYVARS
3727 CONSES
11.193 SECONDS
27.378 SECONDS, REAL TIME
PRETTY
```

If timetyp = 3, time measures and prints garbage collection time as well as computation time, e.g.

```
←TIME((LOAD (QUOTE PRETTY) (QUOTE PROP)) 1 3]
FILE CREATED 7-MAY-71 12:47:14

GC: 8
582, 1091 FREE WORDS
PRETTYFNS
PRETTYVARS
3727 CONSES
10.597 SECONDS
1.487 SECONDS, GARBAGE COLLECTION TIME
PRETTY
```

Another option is timetype=T, in which case time measures and prints the number of pagefaults.

The value of time is the value of the last evaluation of timex.

date[ac3]

If ac3=NIL, obtains date and time from TENEX and returns it as single string in format "dd-mm-yy hh:mm:ss", where dd is day, mm is month, yy year, hh hours, mm minutes, ss seconds, e.g., "14-MAY-71 14:26:08".

Other values of ac3 can be used to specify other formats, e.g. day of week, time zone, etc. as described in TENEX JSYS Manual.

clock[n]

for n=0 current value of the time of day clock i.e., number of milliseconds since last system start up.

for n=1 value of the time of day clock when the user started up this INTERLISP, i.e., difference between clock[0] and clock[1] is number of milliseconds (real time) since this INTERLISP was started.

for n=2 number of milliseconds of *compute* time since user started up this INTERLISP (garbage collection time is subtracted off).

for n=3 number of milliseconds of compute time spent in garbage collections (all types).[1]

dismiss[n]

dismisses program for n milliseconds, during which time program is in a state similar to an I/O wait, i.e., it uses no CPU time. Can be aborted by control-D, control-E, or control-B.

---

[1] This number is directly accessible via the COREVAL GCTIM.

conscount[n] — conscount[] returns the number of conses since INTERLISP started up. If n is not NIL, resets conscount to n.

boxcount[type;n] — number of boxing operations (see Section 13) since INTERLISP started up. If type=NIL, returns number of large integer boxes; type=FLOATING, returns number of floating boxes.[2] If n is not NIL, resets the corresponding counter to n.

gctrp[] — number of conses to next GC: 8, i.e., number of list words not in use. Note that an intervening GC of another type could collect as well as allocate additional list words. See Section 3.

gctrp[n] can be used to cause an interrupt when value of gctrp[]=n, see Section 10.

pagefaults[] — number of page faults since INTERLISP started up.

logout[] — returns to TENEX.[3] A subsequent CONTINUE command will enter the INTERLISP program, return NIL as the value of the call to logout, and continue the computation exactly as if nothing had happened, i.e., logout is a programmable control-C. As with control-C, a REENTER command following a logout will reenter INTERLISP at the top level.

---

[2] These counters are directly accessible via the COREVALs IBOXCN and FBOXCN.

[3] If INTERLISP was started as a subsidiary fork (see subsys, page 21.18), control is returned to the higher fork.

logout[] will not affect the state of any open files.

21.2 Breakdown[4]

Time gives analyses by computation. Breakdown is available to analyze the breakdown of computation time (or any other measureable quantity) function by function. The user calls breakdown giving it a list of functions of interest. These functions are modified so that they keep track of the "charge" assessed to them. The function results gives the analysis of the statistic requested as well as the number of calls to each function. Sample output is shown below.[5]

```
←BREAKDOWN(SUPERPRINT SUBPRINT COMMENT1)
(SUPERPRINT SUBPRINT COMMENT1)
←PRETTYDEF((SUBPRINT) FOO)
(SUBPRINT)
←RESULTS()
FUNCTIONS       TIME       # CALLS
SUPERPRINT      25.294       458
SUBPRINT        32.96        169
COMMENT1        7.833        12
TOTAL           66.087       639
NIL
```

The procedure used for measuring is such that if one function calls other and both are 'broken down', then the time (or whatever quantity is being measured) spent in the inner function is *not* charged to the outer function as well.[6]

---

4 breakdown was written by W. Teitelman.

5 This is with an interpreted prettyprint.

6 breakdown will *not* give accurate results if a function being measured is not returned from normally, e.g. a lower retfrom (or error) bypasses it. In this case, all of the time (or whatever quantity is being measured) between the time that function is entered and the time the next function being measured is entered will be charged to the first function.

To remove functions from those being monitored, simply unbreak the functions, thereby restoring them to their original state. To add functions, call breakdown on the new functions. This will not reset the counters for any functions not on the new list. However breakdown[] can be used for zeroing the counters of all functions being monitored.

To use breakdown for some other statistic, before calling breakdown, set the variable brkdwntype to the quantity of interest, e.g., TIME, CONSES, etc. Whenever breakdown is called with brkdwntype not NIL, breakdown performs the necessary changes to its internal state to conform to the new analysis. In particular, if this is the first time an analysis is being run with this statistic, the compiler may be called to compile the measuring function.[7] When breakdown is through initializing, it sets brkdwntype back to NIL. Subsequent calls to breakdown will measure the new statistic until brkdwntype is again set and a new breakdown performed. Sample output is shown below:

```
←SET(BRKDWNTYPE CONSES)
CONSES
←BREAKDOWN(MATCH CONSTRUCT)
(MATCH CONSTRUCT)
←FLIP((A B C D E F G H C Z) (.. $1 .. #2 ..) (.. #3 ..))
(A B D E F G H Z)
←RESULTS()
FUNCTIONS        CONSES      # CALLS
MATCH            32            1
CONSTRUCT        47            1
TOTAL            79            2
NIL
```

The value of brkdwntype is used to search the list brkdwntypes for the information necessary to analyze this statistic. The entry on brkdwntypes corresponding to brkdwntype should be of the form (type form function), where form computes the statistic, and function (optional) converts the value of form to some more interesting quantity, e.g.

---

[7] The measuring functions for TIME and CONSES have already been compiled.

(TIME (CLOCK 2) (LAMBDA (X) (FQUOTIENT X 1000)))[8] measures computation time and reports the result in seconds instead of milliseconds. If brkdwntype is not defined on brkdwntypes, an error is generated. brkdwntypes currently contains entries for TIME, CONSES, PAGEFAULTS, BOXES, and FBOXES.

More Accurate Measurement

Occasionally, a function being analysed is sufficiently fast that the overhead involved in measuring it obscures the actual time spent in the function. If the user were using time, he would specify a value for timen greater than 1 to give greater accuracy. A similar option is available for breakdown. The user can specify that a function(s) be executed a multiple number of times for each measurement, and the average value reported, by including a number in the list of functions given to breakdown, e.g., BREAKDOWN(EDITCOM EDIT4F 10 EDIT4E EQP) means normal breakdown for editcom and edit4f but executes (the body of) edit4e and eqp 10 times each time they are called. Of course, the functions so measured must not cause any harmful side effects, since they are executed more than once for each call. The printout from results will look the same as though each function were run only once, except that the measurement will be more accurate.

---

[8] For more accurate measurement, the form for TIME is not (CLOCK 2) but (ASSEMBLE NIL (MOVEI 1 , -5) (JSYS 13) (SUB 1 , GCTIM)).

21.3 Edita[9]

Edita is an editor for arrays. However, its most frequent application is in editing compiled functions (which are also arrays in INTERLISP), and a great deal of effort in implementing edita, and most of its special features, are in this area. For example, edita knows the format and conventions of INTERLISP compiled code, and so, in addition to decoding instructions a la DDT,[10] edita can fill in the appropriate COREVALS, symbolic names for index registers, references to literals, linked function calls, etc. The following output shows a sequence of instructions in a compiled function first as they would be printed by DDT, and second by edita.

```
466716/   PUSH 16,LISP&KNIL            3/   PUSH PP,KNIL
466717/   PUSH 16,LISP&KNIL            4/   PUSH PP,KNIL
466720/   HRRZ 1,-12(16)               5/   HRRZ 1,-10(PP)
466721/   CAME 1,LISP&KNIL             6/   CAME 1,KNIL
466722/   JRST 466724                  7/   JRST 9          11
466723/   HRRZ 1,@467575               8/   HRRZ 1,@'BRKFILE
466724/   PUSH 16,1                    9/   PUSH PP,1
466725/   LISP&IOFIL,,467576          10/   PBIND 'BRKZ
466726/   -3,,-3                      11/   -524291
466727/   HRRZ 1,-14(16)              12/   HRRZ 1,-12(PP)
466730/   CAMN 1,467601               13/   CAMN 1,'OK
466731/   JRST 466734                 14/   JRST 17
466732/   CAME 1,467602               15/   CAME 1,'STOP
466733/   JRST 466740                 16/   JRST 21
466734/   PUSH 16,467603              17/   PUSH PP,'BREAK1
466735/   PUSH 16,467604              18/   PUSH PP,'(ERROR!)
466736/   LISP&FILEN,,467605          19/   CCALL 2,'RETEVAL
466737/   JRST 467561                 20/   JRST 422
466740/   CAME 1,467606               21/   CAME 1,'GO
466741/   JRST 466754                 22/   JRST 33
466742/   HRRZ 1,@-12(16)             23/   HRRZ 1,@-10(PP)
466743/   PUSH 16,1                   24/   PUSH PP,1
```

---

[9] edita was written by W. Teitelman.

[10] DDT is one of the oldest debugging systems still around. For users unfamiliar with it, let us simply say that edita was patterned after it because so many people are familiar with it.

[11] Note that edita prints the addresses of cells contained *in* the function relative to the origin of the function.

Therefore, rather than presenting edita as an array editor with some extensions for editing compiled code, we prefer to consider it as a facility for editing compiled code, and point out that it can also be used for editing arbitrary arrays.

Overview

To the user, edita looks very much like DDT with INTERLISP extensions. It is a function of one argument, the name of the function to be edited.[12] Individual registers or cells in the function may be examined by typing their address followed by a slash,[13] e.g.

```
6/   HRRZ 1,-10(PP)
```

The slash is really a command to edita to open the indicated register.[14] Only one register at a time can be open, and *only open registers can be changed.* To change the contents of a register, the user first opens it, types the new contents, and then closes the register with a carriage-return,[15] e.g.

```
7/   CAME1,'↑        CAMN 1,'↑)
```

---

[12] An optional second argument can be a list of commands for edita. These are then executed exactly as though they had come from the teletype.

[13] Underlined characters were typed by the user. edita uses its own read program, so that it is unnecessary to type a space before the slash or to type a carriage return after the slash.

[14] edita also converts absolute addresses of cells within the function to relative address on input. Thus, if the definition of foo begins at 85660, typing 6/ is *exactly* the same as typing 85666/.

[15] Since carriage-return has a special meaning, edita indicates the balancing of parentheses by typing a space.

If the user closes a register without specifying the new contents, the contents are left unchanged. Similarly, if an error occurs or the user types control-E, the open register, if any, is closed without being changed.

Input Protocol

Edita processes all inputs not recognized as commands in the same way. If the input is the name of an instruction (i.e. an atom with a numeric OPD property), the corresponding number is added to the input value being assembled,[16] and a flag is set which specifies that the input context is that of an instruction.

The general form of a machine instruction is (opcode ac , @ address (index)) as described in Section 18. Therefore, in instruction context, edita evaluates all atoms (if the atom has a COREVAL property, the value of the COREVAL is used), and then if the atom corresponds to an ac,[17] shifts it left 23 bits and adds it to the input value, otherwise adds it directly to the input value, but performs the arithmetic in the low 18 bits.[18] Lists are interpreted as specifying index registers, and the value of car of the list (again COREVALs are permitted) is shifted left 18 bits. Examples:

---

[16] The input value is initially 0.

[17] i.e. if a ',' has not been seen, *and* the value of the atom is less than 16, *and* the low 18 bits of the input value are all zero.

[18] If the absolute value of the atom is greater than 1000000Q, full word arithmetic is used. For example, the indirect bit is handled by simply binding @ to 20000000Q.

```
PUSH PP, KNIL
HRRZ 1,-10(PP)
CAME 1, 'GO
JRST 33 ORG
```
[19]

The user can also specify the address of a literal via the ' command, see page 21.14. For example, if the literal " UNBROKEN" is in cell 85672, HRRZ 1,'" UNBROKEN" is equivalent to HRRZ 1, 85672.

When the input context is *not* that of an instruction, i.e. no OPD has been seen, all inputs are evaluated (the value of an atom with a COREVAL property is the COREVAL.) Then numeric values are simply added to the previous input value; non-numeric values *become* the input value.[20]

The only exception to the entire procedure occurs when a register is open that is in the pointer region of the function, i.e. literal table. In this case, atomic inputs are *not* evaluated. For example, the user can change the literal FOO to FIE by simply opening that register and then typing FIE followed by carriage-return, e.g.

<u>'FOO/</u>   FOO   <u>FIE↵</u>

Note that this is equivalent to <u>'FOO/</u> FOO <u>(QUOTE FIE)↵</u>

---

[19] <u>edita</u> cannot in general know whether an address field in an instruction that is typed in is relative or absolute. Therefore, the user must add ORG, the origin of the function, to the address field himself. Note that <u>edita</u> would *print* this instruction, JRST 53 ORG, as JRST 53.

[20] Presumably there is only one input in this case.

↵ (carriage-return) — If a register is open and an input was typed, store the input in the register and close it.[21]

If a register is open and nothing was typed, close the register without changing it.

If a register is not open and input was typed, type its value.

ORG — Has the value of the address of the first instruction in the function. i.e. loc of getd of the function.

/ — Opens the register specified by the low 18 bits of the quantity to the left of the /, and types its contents. If nothing has been typed, it uses the last thing typed by edita, e.g.

```
35/  JRST 53   /   CAME 1,'RETURN    /   RETURN
```

If a register was open, / closes it without changing its contents.

After a / command, edita returns to that state of no input having been typed.

tab (control-I) — Same as carriage-return, followed by the address of the quantity to the left of the tab, e.g.

```
35/  JRST 53   tab
53/  CAME 1,'RETURN
```

Note that if a register was open and input was typed, tab will change the open register before closing it, e.g.

```
35/  JRST 53   JRST 54 tab
54/  JRST 70   ↵
35/  JRST 54
```

. (period) — has the value of the address of the current (last) register examined.

---

[21] If the register is in the unboxed region of the function, the unboxed value is stored in the register.

line-feed	same as carriage-return followed by (ADD1 .)/ i.e. closes any open register and opens the *next* register.
↑	same as carriage-return followed by (SUB1 .)/
$Q (alt-modeQ)	has as its value the last quantity typed by edita e.g.

```
35/  JRST 53   $Q 1↵
./   JRST 54
```

LITS	has as value the (relative) address of the first literal.
BOXED	same as LITS
$ (dollar)	has as value the relative address of the last literal in the function.
=	Sets radix to -8 and types the quantity to the left of the = sign, i.e. if anything has been typed, types the input value, otherwise, types $Q, e.g.

```
35/  JRST 54   =254000241541Q JRST 54=254000000066Q
```

Following =, radix is restored and edita returns to the no input state.

OK	leave edita
?	return to 'no input' state. ? is a 'weak' control-E, i.e. it negates any input typed, but does not close any registers.
address1, address2/	prints[22] the contents of registers address1 through address2. . is set to address2 after the completion.

---

[22] output goes to file, initially set to T. The user can also set file (while in edita) to the name of a disc file to redirect the output. (The user is responsible for opening and closing file.) Note that file only affects output for the address1, address2/ command.

'x corresponds to the ' in LAP. The next expression is read, and if it is a small number, the appropriate offset is added to it. Otherwise, the literal table is searched for x, and the value of 'x is the (absolute) address of that cell. An error is generated if the literal is not found, i.e. ' cannot be used to *create* literals.

:atom defines atom to an address
(1) the value of $Q if a register is open,
(2) the input if any input was typed, otherwise
(3) the value of '.'.[23]

For example:

```
35/  JRST 54   :FOO)
:FIE)
FIE/ JRST FOO  .=35
```

Edita keeps its symbol tables on two free variables, usersyms and symlst. Usersyms is a list of elements of the form (name . value) and is used for *encoding* input, i.e., all variables on usersyms are bound to their corresponding values during evaluation of any expression inside edita. Symlst is a list of elements of the form (value . name) and is used for *decoding* addresses. Usersyms is initially NIL, while symlst is set to a list of all the corevals. Since the : command adds the appropriate information to both these two lists, new definitions will remain in effect even if the user exits from edita and then reenters it later.

Note that the user can effectively define symbols without using the : command by appropriately binding usersyms and/or symlst before calling edita. Also, he can thus use different symbol tables for different applications.

$W (alt-modeW) search command.

Searching consists of comparing the object of the search with the contents of each register, and printing those that match, e.g.

---

[23] Only the low 18 bits are used and converted to a relative address whenever possible.

```
HRRZ @ $W
8/    HRRZ 1,@'BRKFILE
23/   HRRZ 1,@-10(PP)
28/   HRRZ 1,@-12(PP)
```

The $W command can be used to search either the unboxed portion of a function, i.e. instructions, or the pointer region, i.e. literals, depending on whether or not the object of the search is a number. If any input was typed before the $W, it will be the object of the search, otherwise the next expression is read and used as the object.[24] The user can specify a starting point for the search by typing an address followed by a ',' before calling $W, e.g. 1, JRST $W. If no starting point is specified, the search will begin at 0 if the object is a number, otherwise at LITS, the address of the first literal.[25] After the search is completed, '.' is set to the address of the last register that matched.

If the search is operating in the unboxed portion of the function, only those fields (i.e. instruction, ac, indirect, index, and address) of the object that contain one bits are compared.[26] For example, HRRZ @ $W will find all instances of HRRZ indirect, regardless of ac, index, and address fields. Similarly, 'PRINT $W will find all instructions that reference the literal PRINT.[27]

---

[24] Note that inputs typed before the $W will have been processed according to the input protocol, i.e. evaluated; inputs typed after the $W will not. Therefore, the latter form is usually used to specify searching the literals, e.g. $W FOO is equivalent to (QUOTE FOO) $W.

[25] Thus the only way the user can search the pointer region for a number is to specify the starting point via ','.

[26] Alternately, the user can specify his own mask by setting the variable <u>mask</u> (while in <u>edita</u>), to the appropriate bit pattern.

[27] The user may need to establish instruction context for input without giving a specific instruction. For example, suppose the user wants to find all instructions with ac=1 and index=PP. In this case, the user can give & as a pseudo-instruction, e.g. type & 1, (PP).

If the search is operating in the pointer region, a 'match' is as defined in the editor. For example, $W (&) will find all registers that contain a list consisting of a single expression.

$C (alt-modeC) — like $W except only prints the first match, then prints the number of matches when the search finishes.

Editing Arrays

Edita is called to edit a function by giving it the name of the function. Edita can also be called to edit an array by giving it the array as its first argument,[28] in which case the following differences are to be noted:

1. decoding - The contents of registers in the unboxed region are boxed and printed as numbers, i.e. they are never interpreted as instructions, as when editing a function.

2. addressing convention - Whereas 0 corresponds to the first instruction of a function, the first element of an array by convention is element number 1.

3. input protocols - If a register is open, lists are evaluated, atoms are not evaluated (except for $Q which is always evaluated). If no register is open, all inputs are evaluated, and if the value is a number, it is added to the 'input value'.

4. left half - If the left half of an element in the pointer region of an array is not all 0's or NIL, it is printed followed by a ;, e.g.

---

[28] the array itself, *not* a variable whose value is an array, e.g. (EDITA FOO), not EDITA(FOO).

10/ (A B) ; T

Similarly, if a register is closed, either its left half, right half, or both halves can be changed, depending on the presence or absence, and position of the ; e.g.

10/	(A B) ; T	B;⟩	changes left
./	B ; T	NIL⟩	changes right
./	B ; NIL	A ; C⟩	changes both
./	A ; C		

If ; is used in the unboxed portion of an array, an error will be generated.

The SW command will look at both halves of elements in the pointer region, and match if either half matches. Note that SW A ; B is not allowed.

This ends the section on edita.

21.4 Interfork Communication

The functions described below permit two forks (one or both of them INTERLISP) to have a common area of address space for communication by providing a means of assigning a block of storage *guaranteed not to move during garbage collections*.

getblk[n] — Creates a block n *pages* in size (512 words per page). Value is the address of the first word in the block, which is a multiple of 512 since the block will always begin at a page boundary. If not enough pages are available, generates the error ILLEGAL OR IMPOSSIBLE BLOCK.

Note: the block can be used for storing unboxed numbers only.

To store a number in the block, the following function could be used:

```
[SETBLOCK (LAMBDA (START N X) (CLOSER (IPLUS (LOC START) N) X]
```

Some boxing and unboxing can be avoided by making this function compile open via a substitution macro.

Note: getblk should be used sparingly since several unmovable regions of memory can make it difficult or impossible for the garbage collector to find a contiguous region large enough for expanding array space.

relblk[address;n] releases a block of storage beginning at address and extending for n pages. Causes an error ILLEGAL OR IMPOSSIBLE BLOCK if any of the range is not a block. Value is address.

21.5 Subsys[29]

This section describes a function, subsys, which permits the user to run a TENEX subsystem, such as SNDMSG, SRCCOM, TECO, or even another INTERLISP, from inside of an INTERLISP without destroying the latter. In particular, SUBSYS(EXEC) will start up a lower exec, which will print the TENEX herald, followed by @. The user can then do anything at this exec level that he can at the top level, without affecting his superior INTERLISP. For example, he can start another INTERLISP, perform a sysin, run for a while, type a control-C returning him to the lower exec, RESET, do a SNDMSG, etc. The user exits from the lower exec via the command QUIT, which will return control to subsys in the

---

[29] subsys was written by J.W. Goodwin.

higher INTERLISP. Thus with subsys, the user need not perform a sysout to save the state of his INTERLISP in order to use a TENEX capability which would otherwise clobber the core image. Similarly, subsys provides a way of checking out a sysout file in a fresh INTERLISP without having to commandeer another teletype or detach a job.

While subsys can be used to run any TENEX subsystem directly, without going through an intervening exec, this procedure is not recommended. The problem is that control-C always returns control to the next highest *exec*. Thus if the user is running an INTERLISP in which he performs SUBSYS(LISP), and then types control-C to the lower INTERLISP, control will be returned to the exec above the first INTERLISP. The natural REENTER command would then clear the lower INTERLISP,[30] but any files opened by it would remain open (until the next @RESET). If the user elects to call a subsystem directly, he must therefore know how it is normally exited and always exit from it that way.[31]

Starting a lower exec does not have this disadvantage, since it can *only* be exited via QUIT, i.e., the lower exec is effectively 'errorset protected' against control-C.

subsys[file/fork;incomfile;outcomfile;entrypointflg]

If file/fork=EXEC, starts up a lower exec, otherwise runs <SUBSYS>system, e.g. subsys[SNDMSG],subsys[TECO] etc. susbys[] is same as subsys[EXEC]. Control-C always returns control

---

[30] A CONTINUE command however will return to the subordinate program, i.e. control-C followed by CONTINUE is safe at any level.

[31] INTERLISP is exited via the function logout, TECO via the command ;H, SNDMSG via control-Z, and EXEC via QUIT.

to next higher *exec*. Note that more than one INTERLISP can be stacked, but there is no backtrace to help you figure out where you are.

incomfile and outcomfile provide a way of specifying files for input and output. incomfile can also be a string, in which case a temporary file is evaluated, and the string printed on it.

entrypointflg may be START, REENTER, or CONTINUE. NIL is equivalent to START, except when file/fork is a handle (see below) in which case NIL is equivalent to CONTINUE.

The value of subsys is a large integer which is a handle to the lower fork. The lower fork is *not* reset unless the user specifically does so using kfork, described below.[32] If susbys is given as its first argument the value of a previous call to subsys,[33], it continues the subsystem run by that call. For example, the user can do (SETQ SOURCES (SUBSYS TECO)), load up the TECO with a big source file, massage the file, leave TECO with ;H, run INTERLISP for awhile (possibly including other calls to subsys) and then perform (SUBSYS SOURCES) to return to TECO, where he will find his file loaded and even the TECO pointer position preserved.

---

[32] The fork is also reset when the handle is no longer accessible, i.e., when nothing in the INTERLISP system points to it. Note that the fork is accessible while the handle remains on the history list.

[33] Must be the exact same large number, i.e. eq. Note that if the user neglects to set a variable to the value of a call to subsys, (and has performed an intervening call so that sybsys[T] will not work), he can still continue this subsystem by obtaining the value of the call to subsys for the history list using the function valueof, described in Section 22.

Note that if the user starts a lower EXEC, in which he runs an INTERLISP, control-C's from the INTERLISP, then QUIT from the EXEC, if he subsequently continues this EXEC with subsys, he can reenter or continue the INTERLISP.

Note also that calls to subsys can be stacked. For example, using subsys, the user can run a lower INTERLISP, and within that INTERLISP, yet another, etc., and ascend the chain of INTERLISPs using logout, and then descend back down again using subsys.

For convenience, subsys[T] continues the last subsystem run.

SNDMSG, LISP, TECO, and EXEC, are all LISPXMACROS which perform the corresponding calls to subsys. CONTIN is a LISPXMACRO which performs subsys[T], thereby continuing the last subsys.

kfork[fork] — accepts a value from subsys and kills it (RESET in TENEX terminology). If subsys[fork] is subsequently performed, an error is generated. kfork[T] kills all outstanding forks (from this INTERLISP).

21.6 Miscellaneous Tenex Functions[34]

fildir[filegroup] — filegroup is a TENEX file group descriptor, i.e., it can contain stars. fildir returns a list of the files which match filegroup, a la the TENEX DIRECTORY command, e.g. (FILDIR (QUOTE *.COM;0)).

---

[34] All of the functions in section 21.5, except for raise, lowercase, and tenex, were written by J.W. Goodwin.

loadav[ ] returns TENEX current load average as a floating point number (this number is the first of the three printed by the TENEX SYSTAT command).

getab[tablename;index;formatflg] tablename may be any of the tables mentioned with SYSGT and GETAB in JSYS manual; and can be an atom or string.[35] index is the index into the table and must be a number. getab returns the value of that entry of that table as an integer unless formatflg=FLOATING, in which case the value is returned as a floating point number. For example, loadav is defined simply as:
(GETAB (QUOTE SYSTAT) 12 (QUOTE FLOATING)).
Similarly, the host number of the TENEX site you are logged in at may be obtained by (GETAB (QUOTE LHOSTN) 0).

erstr[ern] ern is an error number from a JSYS fail return. ern=NIL means most recent error. erstr returns the TENEX error diagnostic as a string (from <SYSTEM>ERROR.MNEMONICS).

raise[flg] raise[] informs TENEX that it should *not* raise lower case input, i.e., it is equivalent to logout[] @NO RAISE, @CONTINUE. raise[T] informs TENEX that it should raise. raise[T] is the initial setting. The value of raise is the

---

[35] There are some tables in TENEX not documented in the JSYS manual, but getab works for these too if you should happen to hear about them.

previous setting.[36]

lowercase[flg] lowercase[T] performs raise[], sets lcaseflg to flg, and performs other internal modifications to inform DWIM and CLISP that the user is running on a lowercase terminal. lowercase[] reverses the effect of lowercase[T]. The value of lowercase is its previous setting. lowercase is undoable.

lowercase[T] also sets model33flg to NIL (see Section 17). Note that the user can be running on a lowercase terminal and still find it desirable to set model33flg to T if he *thinks* he is on a model 33, i.e. is used to running on one, and makes the typing errors associated with the keyboard layout on a model 33.

usernamela] If a=NIL, returns login directory name; if a=T, returns connected directory name; if a is a number, username returns the user name corresponding to that user number. In all cases, the value is a string.

usernumber[a] If a=NIL, returns login user number; if a=T, returns connected user number; if a is a literal atom or string, usernumber returns the number of the corresponding user, or NIL if no such user exists.

---

[36] raise also sets raiseflg to flg so that the system can determine the raise-noraise state following sysin or reattaching to a detached job, and restore the desired state.

Note: greeting (see Section 22) sets the variable username to the login user name, and firstname to the name used in the greeting.

tenex[str] Starts up a lower EXEC (without a message) using subsys, and then unreads str, followed by "QUIT" (using bksysbuf, described in Section 14). For example, the LISPXMACRO SY which does a SYSTAT is implemented simply as TENEX["SY›"].

21.7 Printing Reentrant and Circular List Structures

A reentrant list structure is one that contains more than one occurrence of the same (eq) structure. For example, tconc (Section 6) makes uses of reentrant list structure so that it does not have to search for the end of the list each time it is called. Thus, if x is a list of 3 elements, (A B C), being constructed by tconc, the reentrant list structure used by tconc for this purpose is:

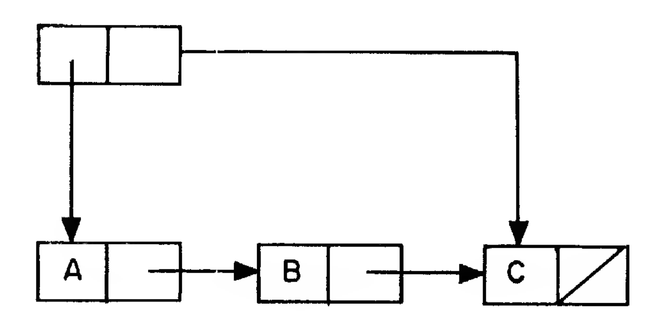


FIGURE 21-1

This structure would be printed by print as ((A B C) C). Note that print would produce the same output for the non-reentrant structure:

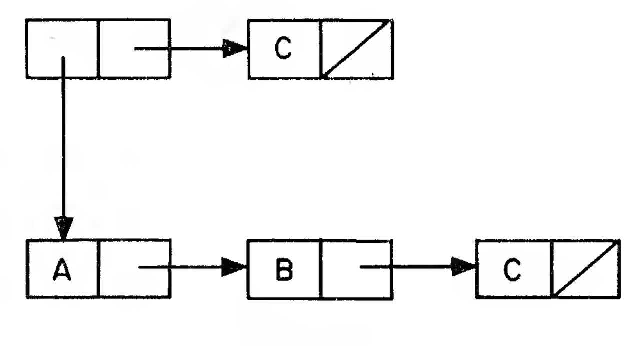


FIGURE 21-2

In other words, print does not indicate the fact that portions of the structure in Figure 21-1 are identical. Similarly, if print is applied to a circular list structure (a special type of reentrant structure) it will never terminate.

For example, if print is called on the structure:

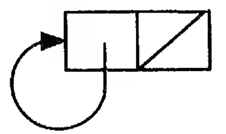


FIGURE 21-3

it will print an endless sequence of left parentheses, and if applied to:

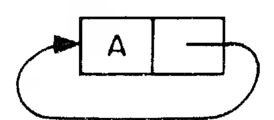


FIGURE 21-4

will print a left parenthesis followed by an endless sequence of A's.

The function circlprint described below produces output that will exactly describe the structure of any circular or reentrant list structure.[37] This output may be in either single or double-line formats. Below are a few examples of the expressions that circlprint would produce to describe the structures discussed above.

```
expression in Figure 21-1:
single-line:  ((A B *1* C) {1})
double-line:  ((A B C) . {1})
                   1
```

```
expression in Figure 21-3:
single-line:  (*1* {1})
double-line:  ({1})
               1
```

```
expression in Figure 21-4:
single-line:  (*1* A . {1})
double-line:  (A . {1})
               1
```

---

[37] Circlprint and circlmaker were written by P.C. Jackson.

The more complex structure:

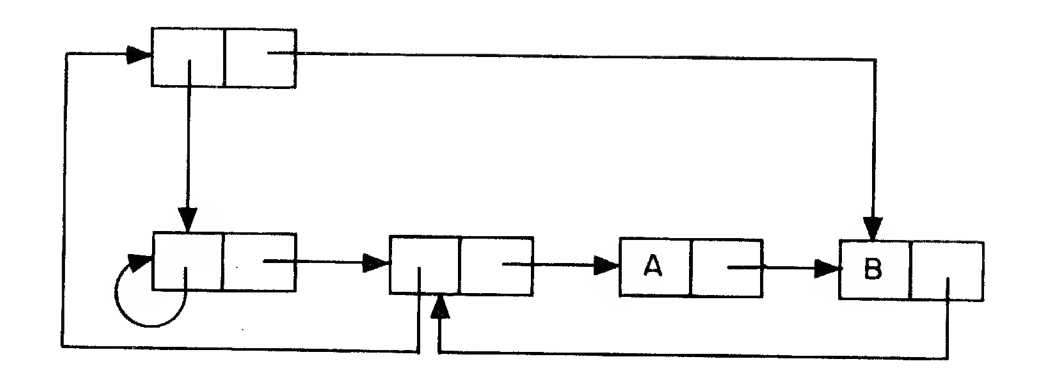


FIGURE 21-5

is printed as follows:

single-line: (*2* (*1* {1} *3* {2} A *4* B . {3}) . {4})

```
double-line:  ( ({1}  {2} A  B . {3}) . {4})
              2 1    3      4
```

In both formats, the reentrant nodes in the list structure are labeled by numbers. (A reentrant node is one that has two or more pointers coming into it.) In the single-line format, the label is printed between asterisks at the beginning of the node (list or tail) that it identifies. In the double-line format, the label is printed below the beginning of the node it identifies. An occurrence of a reentrant node that has already been identified is indicated by printing its label in brackets.

circlprint[list;printflg;rlknt] prints an expression describing list. If printflg=NIL, double-line format is used, otherwise single-line format. circlprint first calls circlmark[list;rlknt], and then calls either rlprin1[list] or rlprin2[list], depending on the value of printflg (T or NIL, respectively). Finally, rlrestore[list] is called, which restores list to its unmarked state. Value is list.

circlmark[list;rlknt]

marks each reentrant node in list with a unique number, starting at rlknt+1 (or 1, if rlknt is NIL). Value is (new) rlknt.

Marking list physically alters it. However, the marking is performed undoably. In addition, list can always be restored by specifically calling rlrestore.

rlprin1[list]

prints an expression describing list in the single-line format. Does not restore list to its uncirclmarked state. list must previously have been circlmarked or an error is generated.

rlprin2[list]

same as rlprin1, except that the expression describing list is printed in the double-line format.

rlrestore[list]

physically restores list to its original, unmarked state.

Note that the user can mark and print several structures which together share common substructures, e.g. several property lists, by making several calls to circlmark, followed by calls to rlprin1 or rlprin2, and finally to rlrestore.

circlmaker[list]

list may contain labels and references following the convention used by circlprint for printing reentrant structures in single line format, e.g. (*1* . {1}). circlmaker performs the necessary rplaca's and rplacd's to make list correspond to the indicated structure. Value is (altered) list.

circlmaker1[list]

Does the work for circlmaker. Uses free variables labelst and reflst. labelst is a list of dotted pairs of labels and corresponding nodes. reflst is a list of nodes containing references to labels not yet seen. Circlmaker operates by initializing labelst and reflst to NIL, and then calling circlmaker1. It generates an error if reflst is not NIL when circlmaker1 returns. The user can call circlmaker1 directly to "connect up" several structures that share common substructures, e.g. several property lists.

Index for Section 21

Page Numbers

SECTION 22

THE PROGRAMMER'S ASSISTANT AND LISPX[1]

22.1 Introduction

This chapter describes one of the newer additions to INTERLISP: the programmer's assistant. The central idea of the programmer's assistant is that the user, rather than talking to a passive system which merely responds to each input and waits for the next, is instead addressing an active intermediary, namely his assistant. Normally, the assistant is invisible to the user, and simply carries out the user's requests. However, since the assistant remembers what the user has told him, the user can instruct him to repeat a particular operation or sequence of operations, with possible modifications, or to undo the effect of certain specified operations. Like DWIM, the programmer's assistant is not implemented as a single function or group of functions, but is instead dispersed throughout much of INTERLISP.[2] Like DWIM, the programmer's assistant embodies a philosophy and approach to system design whose ultimate goal is to construct a programming environment which would "cooperate" with the user in the development of his programs, and free him to concentrate more fully on the conceptual difficulties and creative aspects of the problem he is trying to solve.

---

[1] The programmer's assistant was designed and implemented by W. Teitelman. It is discussed in [Tei4].

[2] Some of the features of the programmer's assistant have been described elsewhere, e.g. the UNDO command in the editor, the file package, etc.

Example

The following dialogue, taken from an actual session at the console, gives the flavor of the programmer's assistant facility in INTERLISP. The user is about to edit a function loadf, which contains several constructs of the form (PUTD FN2 (GETD FN1)). The user plans to replace each of these by equivalent MOVD expressions.

```
←EDITF(LOADFF]                                          [1]
=LOADF
EDIT
*PP
  [LAMBDA (X Y)
    [COND
      ((NULL (GETD (QUOTE READSAVE)))
        (PUTD (QUOTE READSAVE)
              (GETD (QUOTE READ]
    (PUTD (QUOTE READ)
          (GETD (QUOTE REED)))
    (NLSETQ (SETQ X (LOAD X Y)))
    (@UTD (QUOTE READ)
          (GETD (QUOTE READSAVE)))
    X]
*F PUTD (1 MOVD)                                        [2]
*3 (XTRR 2)                                             [3]
=XTR
*0P                                                     [4]
=0 P
(MOVD (QUOTE READSAVE) (QUOTE READ))
*(SW 2 3)                                               [5]
*
```

At [1], the user begins to edit loadf.[3] At [2] the user finds PUTD and replaces it by MOVD. He then shifts context to the third subexpression, [3], extracts its second subexpression, and ascends one level [4] to print and result. The user now switches the second and third subexpression [5], thereby completing

---

[3] We prefer to consider the programmer's assistant as the moving force behind this type of spelling correction (even though the program that does the work is part of the DWIM package). Whereas correcting @PRINT to PRINT, or XTRR to XTR does not require any information about what *this* user is doing, correcting LOADFF to LOADF clearly required noticing when this user defined loadf.

the operation for this PUTD. Note that up to this point, the user has not directly addressed the assistant. The user now requests that the assistant print out the operations that the user has performed, [6], and the user then instructs the assistant to REDO FROM F, [7], meaning repeat the entire sequence of operations 15 through 20. The user then prints the current expression, and observes that the second PUTD has now been successfully transformed.

```
*?? FROM F                                                  [6]

15.  *F PUTD
16.  *(1 MOVD)
17.  *3
18.  *(XTR 2)
19.  *0
20.  *(SW 2 3)

*REDO FROM F                                                [7]
*P
(MOVD (QUOTE REED) (QUOTE READ))
*
```

The user now asks the assistant to replay the last three steps to him, [8]. Note that the entire REDO FROM F operation is now grouped together as a single unit, [9], since it corresponded to a single user request. Therefore, the user can instruct the assistant to carry out the same operation again by simply saying REDO. This time a problem is encountered [10], so the user asks the assistant what it was trying to do [11].

```
*?? FROM -3                                                 [8]

19.  *0
20.  *(SW 2 3)
21. REDO FROM F                                              [9]
     *F PUTD
     *(1 MOVD)
     *3
     *(XTR 2)
     *0
     *(SW 2 3)

*REDO

PUTD ?                                                      [10]

*?? -1                                                      [11]

22.  REDO
```

```
*F PUTD
*(1 MOVD)
*3
*(XTR 2)
*0
```

The user then realizes the problem is that the third PUTD is misspelled in the definition of LOADF (see page 22.2). He therefore instructs the assistant to USE @UTD FOR PUTD, [12], and the operation now concludes successfully.

```
*USE @UTD FOR PUTD                                                    [12]
*P
(MOVD (QUOTE READSAVE) (QUOTE READ))
*↑ PP
   [LAMBDA (X Y)
     [COND
       ((NULL (GETD (QUOTE READSAVE)))
         (MOVD (QUOTE READ)
               (QUOTE READSAVE]
     (MOVD (QUOTE REED)
           (QUOTE READ))
     (NLSETQ (SETQ X (LOAD X Y)))
     (MOVD (QUOTE READSAVE)
           (QUOTE READ))
     X]
*OK
LOADF
←
```

An important point to note here is that while the user *could* have defined a macro to execute this operation, the operation is sufficiently complicated that he would want to try out the individual steps before attempting to combine them. At this point, he would already have executed the operation once. Then he would have to type in the steps again to define them as a macro, at which point the operation would only be repeated once more before failing. Then the user would have to repair the macro, or else change @UTD to PUTD by hand so that his macro would work correctly. It is far more natural to decide *after* trying a series of operations whether or not one wants them repeated or forgotten. In addition, frequently the user will think that the operation(s) in question will never need be repeated, and only discover afterwards that he is mistaken, as occurs when the operation was incorrect, but salvageable:

```
*P
(LAMBDA (STR FLGCQ VRB) **COMMENT** (PROG & & LP1 & LP2 & &))
*-1 -1 P
(RETURN (COND &))
*(-2 ((EQ BB (QUOTE OUT)) BB]                                [1]
*P
(RETURN (& BB) (COND &))                                     [2]
*UNDO
(-2 --) UNDONE
*2 P
(COND (EXPANS & & T))
*REDO EQ
*P
(COND (& BB) (EXPANS & & T)
*
```

Here the operation was correct, [1], but the context in which it was executed, [2], was wrong.

This example also illustrates one of the most useful functions of the programmer's assistant: its UNDO capability. In most systems, if a user suspected that a disaster might result from a particular operation, e.g. an untested program running wild and chewing up a complex data structure, he would prepare for this contingency by saving the state of part or all of his environment before attempting the operation. If anything went wrong, he would then back up and start over. However, saving/dumping operations are usually expensive and time consuming, especially compared to a short computation, and are therefore not performed that frequently, and of course there is always the case when diaster strikes as a result of a 'debugged' or at least innocuous operation, as shown in the following example:

```
←(MAPC ELTS (FUNCTION (LAMBDA (X) (REMPROP X (QUOTE MORPH] [1]
NIL
←UNDO                                                        [2]
MAPC UNDONE.
←USE ELEMENTS FOR ELTS                                       [3]
NIL
←
```

The user types an expression which removes the property MORPH from every member of the list ELTS [1], and then realizes that he meant to remove that property

only from those members of the list ELEMENTS, a much shorter list. In other words, he has deleted a lot of information that he actually wants saved. He therefore simply reverses the effect of the MAPC by typing UNDO [2], and then does what he intended via the USE command [3].

22.2 Overview

The programmer's assistant facility is built around a memory structure called the 'history list.' The history list is a list of the information associated with each of the individual 'events' that have occurred in the system, where each event corresponds to one user input.[4] For example, (XTR 2) ([3] on page 22.2) is a single event, while REDO FROM F ([7] on page 22.3) is also a single event, although the latter includes executing the operation (XTR 2), as well as several others.

Associated with each event on the history list is its input and its value, plus other optional information such as side-effects, formatting information, etc. If the event corresponds to a history command, e.g. REDO FROM F, the input corresponds to what the user would have had to type to execute the same operation(s), although the user's actual input, i.e. the history command, is saved in order to clarify the printout of that event ([9] on page 22.3). Note that if a history command event combines several events, it will have more than one value:

---

[4] For various reasons, there are *two* history lists: one for the editor, and one for lispx, which processes inputs to evalqt and break, see page 22.44.

```
←(LOG (ANTILOG 4))
4.0
←USE 4.0 40 400 FOR 4
4.0
40.0
ARG NOT IN RANGE
400

←USE -40.0 -4.00007 -19.
-40.0
-4.00007
-19.0
←USE LOG ANTILOG FOR ANTILOG LOG IN -2 AND -1
4.0
40.0
400.0
4.00007
19.0
←??

4.  USE LOG ANTILOG FOR ANTILOG LOG IN -2 -1
    ←(ANTILOG (LOG 4.0))
    4.0
    ←(ANTILOG (LOG 40))
    40.0
    ←(ANTILOG (LOG 400))
    400.0
    ←(ANTILOG (LOG -40.0))
    40.0
    ←(ANTILOG (LOG -4.00007))
    4.00007
    ←(ANTILOG (LOG -19.0))
    19.0
3.  USE -40.0 -4.00007 -19.0
    ←(LOG (ANTILOG -40.0))
    -40.0
    ←(LOG (ANTILOG -4.00007))
    -4.00007
    ←(LOG (ANTILOG -19.0))
    -19.0
2.  USE 4.0 40 400 FOR 4
    ←(LOG (ANTILOG 4.0))
    4.0
    (LOG (ANTILOG 40.0)
    40.0
    ←(LOG (ANTILOG 400))

1.  ←(LOG (ANTILOG 4))
    4.0
```

As new events occur, existing events are aged, and the oldest event is 'forgotten.' For efficiency, the storage used to represent the forgotten event is cannibalized and reused in the representation of the new event, so the history list is actually a ring buffer. The size of this ring buffer is a

system parameter called the 'time-slice.'[5] Larger time-slices enable longer 'memory spans,' but tie up correspondingly greater amounts of storage. Since the user seldom needs really 'ancient history,' and a NAME and RETRIEVE facility is provided for saving and remembering selected events, a relatively small time slice such as 30 events is more than adequate, although some users prefer to set the time slice as large as 100 events.

Events on the history list can be referenced in a number of ways. The output on page 22.9 shows a printout of a history list with time-slice 16. The numbers printed at the left of the page are the event numbers. More recent events have higher numbers; the most recent event is event number 52, the oldest and about-to-be-forgotten event is number 37.[6] At this point in time, the user can reference event number 51, RECOMPILE(EDIT), by its event number, 51; its relative position, -2 (because it occurred two events back from the current time), or by a 'description' of its input, e.g. (RECOMPILE (EDIT)), or (& (EDIT)), or even just EDIT. As new events occur, existing events retain their absolute event numbers, although their relative positions change.

Similarly, descriptor references may require more precision to refer to an older event. For example, the description RECOMPILE would have sufficed to refer to event 51 had event 52, also containing a RECOMPILE, not intervened. Event specification will be described in detail later.

---

[5] Initially 30 events. The time-slice can be changed with the function <u>changeslice</u>, page 22.54.

[6] When the event number of the current event is 100, the next event will be given number 1. (If the time slice is greater than 100, the 'roll-over' occurs at the next highest hundred, so that at no time will two events ever have the same event number. For example, if the time slice is 150, event number 1 follows event number 200.)

```
←??

52. ... HIST UNDO
    ←RECOMPILE(HIST)
    HIST.COM
    ←RECOMPILE(UNDO)
    UNDO.COM
51. ←RECOMPILE(EDIT)
    EDIT.COM
50. ←LOGOUT]

49. ←MAKEFILES]
    (EDIT UNDO HIST)
48. ←EDITF(UNDOLISPX)
    UNDOLISPX
47. REDO GETD
    ←GETD(FIE)
    (LAMBDA (X) (MAPC X (F/L (PRINT X))))
46. ←UNDO
    FIE
45. ←GETD(FIE)
    (LAMBDA (X) (MAPC X (FUNCTION (LAMBDA (X) (PRINT X)))))
44. ←FIE]
    NIL
43. ←DEFINEQ((FIE (LAMBDA (X) (MAPC X (F/L (PRINT X))))))
    (FIE)
42. REDO GETD
    ←GETD(FIE)
    (LAMBDA (Y) Y)
41. ←UNDO
    MOVD
40. REDO GETD
    ←GETD(FIE)
    (LAMBDA (X) X)
39. ←MOVD(FOO FIE)
    FIE
38. ←DEFINEQ((FOO (LAMBDA (X) X)))
    (FOO)
37. ←GETD(FIE)
    (LAMBDA (Y) Y)
```

The most common interaction with the programmer's assistant occurs at the top level evalqt, or in a break, where the user types in expressions for evaluation, and sees the values printed out. In this mode, the assistant acts much like a standard LISP evalqt, except that before attempting to evaluate an input, the assistant first stores it in a new entry on the history list. Thus if the operation is aborted or causes an error, the input is still saved and available for modification and/or reexecution. The assistant also notes new functions and variables to be added to its spelling lists to enable future corrections. Then the assistant executes the computation (i.e. evaluates the

form or applies the function to its arguments), saves the value in the entry on the history list corresponding to the input, and prints the result, followed by a prompt character to indicate it is again ready for input.[7]

If the input typed by the user is recognized as a history command, the assistant takes special action. Commands such as UNDO, ??, NAME, and RETRIEVE are immediately performed. Commands that involved reexecution of previous inputs, e.g. REDO and USE, are achieved by computing the corresponding input expression(s) and then *unreading* them. The effect of this unreading operation is to cause the assistant's input routine, lispxread, to act exactly as though these expression were typed in by the user. Except for the fact that these inputs are not saved on new and separate entries on the history list, but associated with the history command that generated them, they are processed exactly as though they had been typed.

The advantage of this implementation is that it makes the programmer's assistant a callable facility for other system packages as well as for users with their own private executives. For example, break1 accept user inputs, recognizes and executes certain break commands and macros, and interprets anything else as INTERLISP expressions for evaluation. To interface break1 with the programmer's assistant required three small modifications to break1: (1) input was to be obtained via lispxread instead of read; (2) instead of calling eval or apply directly, break1 was to give those inputs it could not

---

[7] The function that accepts a user input, saves the input on the history list, performs the indicated computation or history command, and prints the result, is lispx. lispx is called by evalqt and break1, and in most cases, is synonymous with 'programmer's assistant.' However, for various reasons, the editor saves its own inputs on a history list, carries out the requests, i.e. edit commands, and even handles undoing independently of lispx. The editor only calls lispx to execute a history command, such as REDO, USE, etc. Therefore we use the term assistant (loosely) when the discussion applies to features shared by evalqt, break and the editor, and the term lispx when we are discussing the specific function.

interpret to lispx, and (3) any commands or macros handled by break1, i.e. not given to lispx, were to be stored on the history list by break1 by calling the function historysave, a part of the assistant package.

Thus when the user typed in a break command, the command would be stored on the history list as a result of (3). If the user typed in an expression for evaluation, it would be evaluated as before, with the expression and its value both saved on the history list as a result of (2). Now if the user entered a break and typed three inputs: EVAL, (CAR !VALUE), and OK, at the next break, he could achieve the same effect by typing REDO FROM EVAL. This would cause the assistant to unread the three expressions EVAL, (CAR !VALUE), and OK. Because of (1), the next 'input' seen by break1 would then be EVAL, which break1 would interpret. Next would come (CAR !VALUE), which would be given to lispx to evaluate, and then would come OK, which break1 would again process. Thus, by virtue of *unreading*, history operations will work even for those inputs not interpretable by lispx, in this case, EVAL and OK.

The net effect of this implementation of the programmer's assistant is to provide a facility which is easily inserted at many levels, and embodies a consistent set of commands and conventions for talking about past events. This gives the user the subjective feeling that a single agent is watching everything he does and says, and is always available to help.

22.3 Event Specification

All history commands use the same conventions and syntax for indicating which event or events on the history list the command refers to, even though different commands may be concerned with different aspects of the corresponding event(s), e.g. side-effects, value, input, etc. Therefore, before discussing the various history commands in the next section, this section describes the

types of event specifications currently implemented. All examples refer to the history list on page 22.9.

An event address identifies one event on the history list. It consists of a sequence of 'commands' for moving an imaginary cursor up or down the history list, much in the manner of the arguments to the @ command in break (see Section 15). The event identified is the one 'under' the imaginary cursor when there are no more commands. (If any command fails, an error is generated and the history command is aborted.)

The commands are interpreted as follows:

n (n ≥ 1)
: move forward n events, i.e. in direction of increasing event number. If given as the first 'command,' n specifies the event with event number n.

n (n ≤ -1)
: move backward -n events.

←atom
: search backward for an event whose *function* matches atom (i.e. for apply format only), e.g. whereas FIE would refer to event 47, ←FIE would refer to event 44. Similarly, ED\$[8] would specify event 51, whereas ←ED\$ event 48.

pat[9]
: search backward for an event whose input contains an expression that matches pat as described in Section 9.

←
: next search is to go forward instead of backward, (if given as the first 'command', next search begins with last, i.e. oldest, event on history list), e.g. ← LAMBDA refers to event 38; MAKEFILES ← RECOMPILE refers to event 51.

F
: next object is to be searched for, regardless of what it is, e.g. F -2 looks for an event containing a -2.

=
: next search is to look at *values*, instead of

---

[8] i.e. EDalt-mode.

[9] i.e. anything else except for ←, =, and -, which are interpreted as described above.

inputs, e.g. =UNDO refers to event 49; 45 = FIE refers to event 43; ← = LAMBDA refers to event 37.

specifies the event last located.

Note: each search skips the current event, i.e. each command always moves the cursor. For example, if FOO refers to event n, FOO FIE will refer to some event before event n, even if there is a FIE in event n.

An event specification specifies one or more events:

FROM #1 THRU #2
#1 THRU #2
: the sequence of events from the event with address #1 through event with address #2,[10] e.g. FROM GETD THRU 49 specifies events 47, 48, and 49. #1 can be more recent than #2, e.g. FROM 49 THRU GETP specifies events 49, 48, and 47 (note reversal of order).

FROM #1 TO #2
#1 TO #2
: Same as THRU but does not include event #2.

FROM #1
: Same as FROM #1 THRU -1, e.g. FROM 49 specifies events 49, 50, 51, and 52.

THRU #2
: Same as FROM -1 THRU #2, e.g. THRU 49 specifies events 52, 51, 50, and 49. Note reversal of order.

TO
: Same as FROM -1 TO #2.

#1 AND #2 AND ... AND #n
: i.e. a sequence of event addresses separated by AND's, e.g. FROM 47 TO LOGOUT would be equivalent to 47 AND 48 AND MAKEFILES.

empty
: i.e. nothing specified, same as -1, unless last event was an UNDO, in which case same as -2.[11]

---

[10] i.e. the symbol #1 corresponds to all words between FROM and THRU in the event specification, and #2 to all words from THRU to the end of the event specification. For example, in FROM FOO 2 THRU FIE -1, #1 is (FOO 2), and #2 is (FIE -1).

[11] For example, if the user types (NCONC FOO FIE), he can then type UNDO, followed by USE NCONC1.

@ atom — refers to the events named by atom, via the NAME command, page 22.25 e.g., if the user names a particular event or events FOO, @ FOO specifies those events.

@@ ¢ — ¢ is an event specification and interpreted as above, but with respect to the archived history list, as specified on page 22.27.

22.4 History Commands

All history commands can be input as either lists, or as lines (see readline Section 14, and also page 22.47).

¢ is used to denote an event specification. Unless specified otherwise, ¢ omitted is the same as ¢ =-1, e.g. REDO and REDO -1 are the same.

REDO ¢ — redoes the event or events specified by ¢, e.g. REDO FROM -3 redoes the last three events.

USE vars FOR args IN ¢ — substitutes vars for args in ¢, and redoes the result, e.g.
USE LOG ANTILOG FOR ANTILOG LOG IN -2 AND -1.
Substitution is done by esubst, Section 9, and is carried out as described below.

USE $vars_1$ FOR $args_1$ AND ... AND $vars_n$ FOR $args_n$ IN ¢ — More general form of USE command. See description of substitution algorithm below.

Every USE command involves three pieces of information: the variables to be substituted, the arguments to be substituted for, and an event specification, which defines the expression (input) in which the substitution takes place.[12]

If args are omitted, i.e. the form of the command is USE vars IN ¢, or just USE vars (which is equivalent to USE vars IN -1), and the event referred to was itself a USE command, the arguments and expression substituted into are the same as for the indicated USE command. In effect, this USE command is thus a

---

[12] The USE command is parsed by a small finite state parser to distinguish the variables and arguments. For example, USE FOR FOR AND AND AND FOR FOR will be parsed correctly.

continuation of the previous USE command. For example, on page 22.7, when the user types (LOG (ANTILOG 4)), followed by USE 4.0 40 400 FOR 4, followed by USE -40.0 -4.00007 -19., the latter command is equivalent to USE -40.0 -4.00007 -19. FOR 4 IN -2.

If args are omitted and the event referred to was *not* a USE command, substitution is for the operator in that command, i.e. if a lispx input, the name of the function, if an edit command, the name of the command. For example ARGLIST(FF) followed by USE CALLS is equivalent to USE CALLS FOR ARGLIST.

If ¢ is omitted, but args are specified, the first member of args is used for ¢, e.g. USE PUTD FOR @UTD is equivalent to USE PUTD FOR @UTD IN F @UTD.[13]

If the USE command has the same number of expressions as arguments, the substitution procedure is straightforward,[14] i.e. USE X Y FOR U V means substitute X for U and Y for V, and is equivalent to USE X FOR U AND Y FOR V. However, the USE command also permits distributive substitutions, i.e. substituting several expressions for the same argument. For example, USE A B C FOR X means first substitute A for X then substitute B for X (in a new copy of the expression), then substitute C for X. The effect is the same as three separate USE commands. Similarly, USE A B C FOR D AND X Y Z FOR W is equivalent to USE A FOR D AND X FOR W, followed by USE B FOR D AND Y FOR W,

---

[13] The F is inserted to handle correctly the case where the first member of args is a number, e.g. USE 4.0 4.0 400 FOR 4. Obviously the user means find the event containing a 4 and perform the indicated substitutions, whereas USE 4.0 40 400 FOR 4 IN 4 would mean perform the substitutions in event *number* 4.

[14] Except when one of the arguments and one of the variables are the same, e.g. USE X Y FOR Y X, or USE X FOR Y AND Y FOR X. This situation is noticed when parsing the command, and handled correctly.

followed by USE C FOR D AND Z FOR W. USE A B C FOR D AND X FOR Y[15] also corresponds to three substitions, the first with A for D and X for Y, the second with B for D, and X for Y, and the third with C for D, and again X for Y. However, USE A B C FOR D AND X Y FOR Z is ambiguous and will cause an error. Essentially, the USE command operates by proceeding from left to right handling each 'AND' separately. Whenever the number of expressions exceeds the number of expressions available, the expressions multiply.[16]

FIX ¢ — puts the user in the editor looking at a copy of the input(s) for ¢, Whenever the user exits via, OK, the result is unread and reexecuted exactly as with REDO.

FIX is provided for those cases when the modifications to the input(s) are not of the type that can be specified by USE, i.e. not substitutions. For example:

```
←(DEFINEQ FOO (LAMBDA (X) (FIXSPELL SPELLINGS2 X 70]

INCORRECT DEFINING FORM
FOO

←FIX
EDIT
*p
(DEFINEQ FOO (LAMBDA & &))
*(LI 2)
OK
(FOO)
←
```

The user can also specify the edit command(s) to lispx, by typing - followed by the command(s) after the event specification, e.g. FIX - (LI 2). In this case, the editor will not type EDIT, or wait for an OK after executing the commands.

---

15 or USE X FOR Y AND A B C FOR D.

16 Thus USE A B C D FOR E F means substitute A for E at the same time as substituting B for F, then in another copy of the indicated expression, substitute C for E and D for F. Note that this is also equivalent to USE A C FOR E AND B D FOR F.

Implementation of REDO, USE, and FIX

The input portion of an event is represented internally on the history list simply as a linear sequence of the expressions which were read. For example, an input in apply format is a list consisting of two expressions, an an input in eval format is a list of just one expression.[17] Thus if the user wishes to convert an input in apply format to eval format, he simply moves the function name inside of the argument list:

```
←MAPC(FOOFNS (F/L (AND (EXPRP X) (PRINT X]
NIL
←EXPRP(FOO1)
T
←FIX MAPC
EDIT
*P
(MAPC (FOOFNS &))
*(BO 2)
*(LI 1)
*P
((MAPC FOOFNS &))
OK
FOO1
FIE2
FUM
NIL
←
```

By simply converting the input from two expressions to one expression, the desired effect, that of mapping down the list that was the *value* of foofns, was achieved.

REDO, USE, and FIX all operate by obtaining the input portion of the corresponding event, processing the input (except for REDO), and then storing

---

17 For inputs in eval format, i.e. single expressions, FIX calls the editor so that the current expression is that input, rather than the list consisting of that input - see the example on the preceding page. However, the entire list is actually being edited. Thus if the user typed ↑ P in that example, he would see ((DEFINEQ FOO &)).

it on the history list as the input portion of a new event. The history command completes operating by simply unreading the input. When the input is subsequently 'reread', the event which already contains the input will be retrieved and used for recording the value of the operation, saving side-effects, etc., instead of creating a new event. Otherwise the input is treated exactly the same as if it had been typed in directly.

If ¢ specifies more than one event, the inputs for the corresponding events are simply concatenated into a linear sequence, with special markers representing carriage returns[18] inserted between each input to indicate where new lines start. The result of this concatenation is then treated as the input referred to by ¢. For example, when the user typed REDO FROM F ([7] on page 22.3) the inputs for the corresponding six events were concatenated to produce (F PUTD #0 (1 MOVD) #0 3 #0 (XTR 2) #0 0 #0 (SW 2 3)). Similarly, if the user had typed USE @UTD FOR PUTD IN 15 THRU 20, (F PUTD #0 (1 MOVD) #0 3 #0 (XTR 2) #0 0 #0 (SW 2 3)) would have been constructed, and then @UTD substituted for PUTD throughout it.

The same convention is used for representing multiple inputs when a USE command involves sequential substitutions. For example, if the user types GETD(FOO) and then USE FIE FUM FOR FOO, the input sequence that will be constructed is (GETD (FIE) #0 GETD (FUM)), which is the result of substituting FIE for FOO in (GETD (FOO)) concatenated with the result of substituting FUM for FOO in (GETD (FOO)).

Once such a multiple input is constructed, it is treated exactly the same as a single input, i.e. the input sequence is recorded in a new event, and then

---

18 The value of (VAG 0) is currently used to represent a carriage return on the grounds that it cannot be typed in by the user, and thus cannot cause ambiguities.

unread, exactly as described above. When the inputs are 'reread,' the 'pseudo-carriage-returns' are treated by lispxread and readline exactly as real carriage returns, i.e. they serve to distinguish between apply and eval formats on inputs to lispx, and to delimit line commands to the editor. Note that once this multiple input has been entered as the input portion of a new event, that event can be treated exactly the same as one resulting from type in. In other words, no special checks have to be made when *referencing* an event, to see if it is simple or multiple. Thus, when the user types REDO following REDO FROM F, ([10] page 22.3) REDO does not even notice that the input retrieved from the previous event is (F PUTD #0 ... (SW 2 3)) i.e. a multiple input, it simply records this input and unreads it. Similarly, when the user then types USE @UTD FOR PUTD on this multiple input, the USE command simply carries out the substitution, and the result is the same as though the user had typed USE @UTD FOR PUTD IN 15 THRU 20.

In sum, this implementation permits ¢ to refer to a single simple event, or to several events, or to a single event originally constructed from several events (which may themselves have been multiple input events, etc.) without having to treat each case separately.

History Commands Applied to History Commands

Since history commands themselves do *not* appear in the input portion of events (although they are stored elsewhere in the event), they do not interfere with or affect the searching operations of event specifications. In effect, history commands are invisible to event specifications.[19] As a result, history commands themselves cannot be recovered for execution in the normal way. For example,

---

[19] With the exception described below under "History Commands that Fail".

if the user types USE A B C FOR D and follows this with USE E FOR D, he will not produce the effect of USE A B C FOR E (but instead will simply cause E to be substituted for D in the last event containing a D). To produce this effect, i.e. USE A B C FOR E, the user should type USE D FOR E IN *USE*. The appearance of the word REDO, USE or FIX in an event address specifies a search for the corresponding *history* command. (For example, the user can also type UNDO REDO.) It also specifies that the text of the history command itself be treated as though it were the input. However, *the user must remember that the context in which a history command is reexecuted is that of the current history, not the original context.* For example, if the user types USE FOO FOR FIE IN -1, and then later types REDO USE, the -1 will refer to the event before the REDO, not before the USE. Similarly, if the user types REDO REDO followed by REDO REDO, he would cause an infinite loop, except for the fact that a special check detects this type of situation.

History Commands that Fail

The one exception to the statement that 'history commands are invisible to event specifications' occurs when a history command fails to produce any input. For example, suppose the user types USE LOG FOR ANTILOG AND ANTILOG FOR LOGG, causing lispx to respond LOGG ?. Since the USE command did not produce any input, the user can repair it by typing USE LOG FOR LOGG (i.e. does not have to specify IN USE). This latter USE command will invoke a search for LOGG, which *will* find the bad USE command. lispx then performs the indicated substitution, and unreads USE LOG FOR ANTILOG AND ANTILOG FOR LOG. In turn, this USE command invokes a search for ANTILOG, which, *because it was not typed in but reread*, ignores the bad USE command which was found by the earlier search for LOGG, and which is still on the history list. In other words, *history commands that fail to produce input are visible to searches arising from event specifications typed in by the user, but not to secondary event specifications.*

In addition, if the most recent event is a history command which failed to produce input, a secondary event specification will effectively back up the history list one event so that relative event numbers for that event specification will not count the bad history command. For example, suppose the user types USE LOG FOR ANTILOG AND ANTILOG FOR LOGG IN -2 AND -1, and after lispx types LOGG ?, the user types USE LOG FOR LOGG. He thus causes the command USE LOG FOR ANTILOG AND ANTILOG FOR LOG IN -2 AND -1 to be constructed and unread. In the normal case, -1 would refer to the last event, i.e. the 'bad' USE command, and -2 to the event before it. However, in this case, -1 refers to the event before the bad USE command, and the -2 to the event before that. In short, the caveat that "the user must remember that the context in which a history command is reexecuted is that of the current history, not the original context" does not apply if the correction is performed immediately.

More History Commands

RETRY ¢ — similar to REDO except sets helpclock so that any errors that occur while executing ¢ will cause breaks.

... vars — similar to USE except substitutes for the (first) *operand*.

For example, EXPRP(FOO) followed by ... FIE FUM is equivalent to USE FIE FUM FOR FOO. See also event 52 on page 22.9.

?? ¢ — prints history list. If ¢ is omitted, ?? prints the entire history list, beginning with most recent events. Otherwise ?? prints only those events specified in ¢ (and in the order specified), e.g. ?? -1, ?? 10 THRU 15, etc.

?? commands are not entered on the history list, and so do not affect relative event numbers. In other words, an event specification of -1 typed following a ?? command will refer to the event immediately preceding the ?? command.

?? will print the history command, if any, associated with each event as shown at [9] on page 22.3 and page 22.7. Note that these history commands are not preceded by prompt characters, indicating they are not stored as input.[20]

?? prints multiple input events under one event number (see page 22.7).

Since events are initially stored on the history list with their value field equal to bell (control-G), if an operation fails to complete for any reason, e.g. causes an error, is aborted, etc., its 'value' will be bell. This is the explanation for the blank line in event 2, page 22.7, and event 50, page 22.9.

?? is implemented via the function printhistory, page 22.60, which can also be called directly by the user.

* * *

UNDO ¢ — undoes the side effects of the specified events. For each event undone, UNDO prints a message: e.g. RPLACA UNDONE, REDO UNDONE etc. If nothing is undone because nothing was saved, UNDO types NOTHING SAVED. If nothing was undone because the event(s) were already undone, UNDO types ALREADY UNDONE. If ¢ is empty, UNDO searches back for the last event that contained side effects, was not undone, and itself was not an UNDO

---

[20] REDO, RETRY, USE, ..., and FIX commands, i.e. those commands that reexecute previous events, are not stored as inputs, because the input portion for these events are the expressions to be 'reread'. The history commands UNDO, NAME, RETRIEVE, BEFORE, and AFTER *are* recorded as inputs, and ?? prints them exactly as they were typed.

command.[21] [22]

UNDO ¢ : x_1 ... x_n — Each x_i refers to a message printed by DWIM in the event(s) specified by ¢. The side effects of the corresponding DWIM corrections, and only those side effects, are undone.

For example, if the message PRINTT [IN FOO] -> PRINT were printed, UNDO : PRINTT or UNDO : PRINT would undo the correction.[23]

* * *

$ is a special form of the USE command for conveniently specifying *character* substitutions. In addition, it has a number of useful default options in connection with events that involve errors.

$ x FOR y — equivalent to USE x FOR y

For example, the user types MOVD(FOO FOOSAVE T), he can then type $ FIE FOR FOO to perform MOVD(FIE FIESAVE T). Note that USE FIE FOR FOO would perform MOVD(FIE FOOSAVE T).

---

[21] Note that the user can undo UNDO commands themselves by specifying the corresponding event address, e.g. UNDO -3 or UNDO UNDO.

[22] UNDOing events in the reverse order from which they were executed is guaranteed to restore all pointers correctly, e.g. to undo all effects of last five events, perform UNDO THRU -5, *not* UNDO FROM -5. Undoing out of order may have unforseen effects if the operations are *dependent*. For example, if the user performed (NCONC1 FOO FIE), followed by (NCONC1 FOO FUM), and then undoes the (NCONC1 FOO FIE), he will also have undone the (NCONC1 FOO FUM). If he then undoes the (NCONC1 FOO FUM), he will cause the FIE to reappear, by virtue of restoring FOO to its state before the execution of (NCONC1 FOO FUM). For more details, see page 22.42.

[23] Some portions of the messages printed by DWIM are strings, e.g. the message FOO UNSAVED is printed by printing FOO and then " UNSAVED". Therefore, if the user types UNDO : UNSAVED, the DWIM correction will not be found. He should instead type UNDO : FOO or UNDO : $UNSAVED$ (alt-modeUNSAVEDalt-mode, see R command in editor, section 9).

An abbreviated form of $ is available:

$ y x	same as $ x FOR y, i.e. y's are changed to x's. can also be written as $ y TO x, $ y = x, or $ y -> x.

$ does event location the same as the USE command, i.e. if IN -- is not specified, it searches for y.[24]

After $ finds the event, it looks to see if an error was involved in that event,[25] and if the indicated character substitution can be performed in the offender. If so, $ assumes the substitution refers to the offender, performs the substitution, and then substitutes the result for the offender throughout. For example, the user types (PRETTYDEF FOOFNS 'FOO FOOVARS) causing a U.B.A. FOOOVARS error message. The user can now type $ OO O, which will change FOOOVARS to FOOVARS, but *not* change FOOFNS or FOO.

If an error did occur in the specified event, the user can also omit specifying y, in which case the offender is used. Thus, the user could have corrected the above example by simply typing $ FOOVARS. Similarly, if the user types LOAD(PRSTRUC PROP), causing the error FILE NOT FOUND PRSTRUC, he can request the file to be loaded from LISP's directory by simply typing $ <LISP>$. Since esubst is used for substituting, this is equivalent to performing (R PRSTRUC <LISP>$) on the event, and therefore replaces PRSTRUC by <LISP>PRSTRUC (see Section 9). Note also the usefulness of $ '$, meaning: put a ' in front of the offender.

---

[24] However, unlike USE, $ can only be used to specify one substitution at a time.

[25] Whenever an error occurs, the object of the error message, called the offender, is automatically saved on that event's entry in the history list, under the property ERROR.

$ also works for events in the editor. For example, if the user types (MOVE COND 33 2 TO BEFORE HERE), and editor types 33 ?, the user can type $ 3, causing 3 to be substituted for 33 in the MOVE command.

Finally, the user can omit both x and y. This specifies that two alt-modes be packed onto the end of the offender, and the result substituted throughout the specified event. For example, suppose the user types to the editor (MOVE 3 2 TO AFTER CONDD 1), and gets the error message CONDD ?. because the find command failed to find CONDD. $ will cause the edit command (MOVE 3 2 TO AFTER CONDD$$ 1) to be executed, which will search for an atom that is "close" to CONDD in the sense used by the spelling corrector (see pattern type 6b, Section 9).[26]

Note that $ never *searches* for an error. Thus, if the user types LOAD(PRSTRUC PROP) causing a FILE NOT FOUND error, types CLOSEALL(), and *then* types $ <LISP>$, lispx will complain that there is no error in CLOSEALL(). In this case, the user would have to type $ <LISP>$ IN LOAD, or $ PRS <LISP>PRS (which would cause a search for PRS).

Note also that $ operates on *input*, not on programs. If the user types FOO(), and within the call to FOO gets a U.D.F. CONDD error, he *cannot* repair this by $ COND. lispx will type CONDD NOT FOUND IN FOO().

* * *

NAME atom ¢ — saves the event(s) (including side effects) specified by ¢ on the property list of atom (under the property HISTORY) e.g. NAME FOO 10 THRU 15. NAME commands are undoable.

---

[26] The same effect could be achieved by $ COND, which specifies substituting COND for CONDD, but *not* by $ CONDD$$, since the latter is equivalent to performing (R CONDD CONDD$$) on the event, which would result in CONDDCONDDCONDD being substituted for CONDD (as described in Section 9).

RETRIEVE atom — Retrieves and reenters on the history list the events named by atom. Causes an error if atom was not named by a NAME command.

For example, if the user performs NAME FOO 10 THRU 15, and at some time later types RETRIEVE FOO, 6 *new* events will be recorded on the history list (whether or not the corresponding events have been forgotten yet). Note that RETRIEVE does *not* reexecute the events, it simply retrieves them. The user can then REDO, UNDO, FIX, etc. any or all of these events. Note that the user can combine the effects of a RETRIEVE and a subsequent history command in a single operation by using an event specification of the form @ atom, as described on page 22.14, e.g. REDO @ FOO is equivalent to RETRIEVE FOO, followed by an appropriate REDO.[27] Note that UNDO @ FOO and ?? @ FOO are permitted.

BEFORE atom — undoes the effects of the events named by atom.

AFTER atom — undoes a BEFORE atom.

BEFORE/AFTER provide a convenient way of flipping back and forth between two states, namely that state *before* a specified event or events were executed, and that state *after* execution. For example, if the user has a complex data structure which he wants to be able to interrogate before and after certain modifications, he can execute the modifications, name the corresponding events with the NAME command, and then can turn these modifications off and on via BEFORE or AFTER commands.[28] Both BEFORE and AFTER are NOPs if the atom was

---

[27] Actually, REDO @ FOO is better than RETRIEVE followed by REDO since in the latter case, the corresponding events would be entered on the history list *twice*, once for the RETRIEVE and once for the REDO.

[28] The alternative to BEFORE/AFTER for repeated switching back and forth involves UNDO, UNDO of the UNDO, UNDO of that etc. At each stage, the user would have to locate the correct event to undo, and furthermore would run the risk of that event being 'forgotten' if he did not switch at least once per time-slice.

already in the corresponding state; both generate errors if atom was not named by a NAME command.

Note: since UNDO, NAME, RETRIEVE, BEFORE, and AFTER are recorded as inputs they can be referenced by REDO, USE, etc. in the normal way. However, the user must again remember that the context in which the command is reexecuted is different than the original context. For example, if the user types NAME FOO DEFINEQ THRU COMPILE, then types ... FIE, the input that will be reread will be NAME FIE DEFINEQ THRU COMPILE as was intended, but both DEFINEQ and COMPILE, will refer to the most recent event containing those atoms, namely the event consisting of NAME FOO DEFINEQ THRU COMPILE!

ARCHIVE ¢ — records the events specified by ¢ on a permanent history list. This history list can be referenced by preceding a standard event specification with @@, e.g. ?? @@ prints the archived history list, REDO @@ -1 will recover the corresponding event from the archived history list and redo it, etc.

The user can also provide for automatic archiving of selected events by appropriately defining archivefn, as described on page 22.33.

FORGET ¢ — permanently erases the record of the side effects for the events specified by ¢. If ¢ is omitted, forgets side effects for entire history list.

FORGET is provided for users with space problems. For example, if the user has just performed sets, rplacas, rplacds, putd, remprops, etc. to release storage, the old pointers would not be garbage collected until the corresponding events age sufficiently to drop off the end of the history list and be forgotten. FORGET can be used to force immediate forgetting (of the side-effects only). FORGET is not undoable (obviously).

22.5 Miscellaneous Features and Commands

TYPE-AHEAD is a command that allows the user to type-ahead an indefinite number of inputs.

The assistant responds to TYPE-AHEAD with a prompt character of >. The user can now type in an indefinite number of lines of input, under errorset protection. The input lines are saved and unread when the user exits the type-ahead loop with the command $GO (alt-modeGO). While in the type-ahead loop, ?? can be used to print the type-ahead, FIX to edit the type-ahead, and $Q to erase the last input (may be used repeatedly). For example:

```
←TYPE-AHEAD
>SYSOUT(TEM)
>MAKEFILE(EDIT)
>BRECOMPILE((EDIT WEDIT))
>F
>$Q
\\F
>$Q
\\BRECOMPILE
>LOAD(WEDIT PROP)
>BRECOMPILE((EDIT WEDIT))
>F
>MAKEFILE(BREAK)
>LISTFILES(EDIT BREAK)
>SYSOUT(CURRENT)
>LOGOUT]
>??
     >SYSOUT(TEM)
     >MAKEFILE(EDIT)
     >LOAD(WEDIT PROP)
     >BRECOMPILE((EDIT WEDIT))
     >F                                  29
     >MAKEFILE(BREAK)
     >LISTFILES(EDIT BREAK)
     >SYSOUT(CURRENT)
     >LOGOUT]
>FIX
EDIT
*(R BRECOMPILE BCOMPL)
*P
((LOGOUT) (SYSOUT &) (LISTFILES &) (MAKEFILE &) (F) (BCOMPL &)
(LOAD &) (MAKEFILE &) (SYSOUT &))
*(DELETE LOAD)
*OK
>$GO
```

The TYPE-AHEAD command may be aborted by $STOP; control-E simply aborts the current line of input.

---

[29] Note that type-ahead can be addressed to the compiler, since it uses <u>lispxread</u> for input. Type-ahead can also be directed to the editor, but type-ahead to the editor and to <u>lispx</u> cannot be intermixed.

* * *

$BUFS (alt-modeBUFS) is a command for recovering the input buffers.

Whenever an error occurs in executing a lispx input or edit command, or a control-E or control-D is typed, the input buffers are saved and cleared. The $BUFS command is used to restore the input buffers, i.e. its effect is exactly the same as though the user had retyped what was 'lost.' For example:

```
*(-2 (SETQ X (COND ((NULL Z) (CONS     (user typed control-E)
*P
(COND (& &) (T &))
*2
*$BUFS
(-2 (SETQ X (COND ((NULL Z) (CONS
```

and user can now finish typing the (-2 ..) command.

Note: the type-ahead does not have to have been seen by INTERLISP, i.e., echoed, since the system buffer is also saved.

Input buffers are not saved on the history list, but on a free variable. Thus, only the contents of the input buffer as of the last clearbuf can ever be recovered. However, input buffers cleared at evalqt are saved independently from those cleared by break or the editor. The procedure followed when the user types $BUFS is to recover first from the local buffer, otherwise from the top level buffer.[30] Thus the user can lose input in the editor, go back to evalqt, lose input there, then go back into the editor, recover the editor's

---

[30] The local buffer is stored on lispxbufs; the top level buffer on toplispxbufs. The forms of both buffers are (CONS (LINBUF) (SYSBUF)) (see Section 14). Recovery of a buffer is destructive, i.e. $BUFS sets the corresponding variable to NIL. If the user types $BUFS when both lispxbufs and toplispxbufs are NIL, the message NOTHING SAVED is typed, and an error generated.

buffer, etc. Furthermore, a buffer cleared at the top can be recovered in a break, and vice versa.

* * *

The following four commands, DO, !F, !E, and !N, are only recognized in the editor:

DO com — allows the user to supply the command name when it was omitted. (USE is used when a command name is *incorrect*).

For example, suppose the user wants to perform (-2 (SETQ X (LIST Y Z))) but instead types just (SETQ X (LIST Y Z)). The editor will type SETQ ?, whereupon the user can type DO -2. The effect is the same as though the user had typed FIX, followed by (LI 1), (-1 -2), and OK, i.e. the command (-2 (SETQ X (LIST Y Z))) is executed. DO also works if the last command is a line command.

!F — same as DO F.

In the case of !F, the previous command is always treated as though it were a line command, e.g. if the user types (SETQ X &) and then !F, the effect is the same as though he had typed F (SETQ X &), not (F (SETQ X &)).

!E — same as DO E. Note !E works correctly for 'commands' typed in <u>eval</u> or <u>apply</u> format.

!N — same as DO N.

*   *   *

control-U — when typed in at any point during an input being read by lispxread, permits the user to edit the input before it is returned to the calling function.

This feature is useful for correcting mistakes noticed in typing *before* the input is executed, instead of waiting till after execution and then performing an UNDO and a FIX. For example, if the user types (DEFINEQ (FOO (LAMBDA (X) (FIXSPELL X and at that point notices the missing left parenthesis, instead of completing the input and allowing the error to occur, and then fixing the input, he can simply type control-U,[31] finish typing normally, whereupon the editor is called on (FOO (LAMBDA (X) (FIXSPELL X --], which the user can then repair, e.g. by typing (LI 1). If the user exits from the editor via OK, the (corrected) expression will be returned to whoever called lispxread exactly as though it had been typed.[32] If the user exits via STOP, the expression is returned so that it can be stored on the history list. However it will *not* be executed. In other words, the effect is the same as though the user had typed control-E at exactly the right instant.

---

[31] Control-U can be typed at any point, even in the middle of an atom; it simply sets an internal flag checked by lispxread.

[32] Control-U also works for calls to readline, i.e., for line commands.

*   *   *

valueof

is an nlambda function for obtaining the value of a particular event, e.g. (VALUEOF -1), (VALUEOF ←FOO -2).[33]

The value of an event consisting of several operations is a list of the values for each of the individual operations.

Note: the value field of a history entry is initialized to bell (control-G). Thus a value of bell indicates that the corresponding operation did not complete, i.e. was aborted or caused an error (or else returned bell).

*   *   *

prompt#flg

is a flag which when set to T causes the current event number to be printed before each ←,: and * prompt characters. See description of promptchar, page 22.51.

prompt#flg is initially NIL.

*   *   *

archivefn

allows the user to specify events to be automatically archived.

When archivefn is *set* to T, and an event is about to drop off the end of the history list and be forgotten, archivefn is called giving it as its first argument the input portion of the event, and as its second argument, the entire

---

[33] Although the input for valueof is entered on the history list before valueof is called, valueof[-1] still refers to the value of the expression immediately before the valueof input, because valueof effectively backs the history list up one entry when it retrieves the specified event. Similarly, (VALUEOF FOO) will find the first event before this one that contains a FOO.

event.[34] If archivefn returns T, the event is archived. For example, some users like to keep a record of all calls to load. Defining archivefn as: (LAMBDA (X Y) (EQ (CAR X) (QUOTE LOAD))) will accomplish this. Note that archivefn must be *both* set and defined. archivefn is initially NIL and undefined.

* * *

lispxmacros provides a macro facility for lispx.

lispxmacros allows the user to define his own lispx commands. It is a list of elements of the form (command def). Whenever command appears as the first expression on a line in a lispx input, the variable lispxline is bound to the rest of the line, the event is recorded on the history list, and def is evaluated. Similarly, whenever command appears as car of a form in a lispx input, the variable lispxline is bound to cdr of the form, the event recorded, and def is evaluated. (See page 22.60 for an example of a lispxmacro). RETRIEVE, BEFORE, and AFTER are implemented as lispxmacros. In addition, LISP, SNDMSG, TECO, and EXEC are lispxmacros which perform the corresponding calls to subsys (section 21), and CONTIN is a lispxmacro which performs (SUBSYS T). Finally, SY and DIR are lispxmacros which perform the EXEC, SYSTAT, and DIRECTORY commands respectively. DIR can be given arguments, e.g., DIR *.SAV;*.

---

[34] In case archivefn needs to examine the value of the event, its side effects, etc. See page 22.44 for discussion of the format of history lists.

*   *   *

lispxuserfn — provides a way for a user function to process selected inputs.

When lispxuserfn is set to T, it is applied[35] to all inputs not recognized as one of the commands described above. If lispxuserfn decides to handle this input, it simply processes it (the event was already stored on the history list before lispxuserfn was called), sets lispxvalue to the value for the event, and returns T. lispx will then know not to call eval or apply, and will simply store lispxvalue into the value slot for the event, and print it. If lispxuserfn returns NIL, lispx proceeds by calling eval or apply in the usual way. Thus by appropriately defining (and setting) lispxuserfn, the user can with a minimum of effort incorporate the features of the programmer's assistant into his own executive (actually it is the other way around).

---

[35] Like archivefn, lispxuserfn must be both set and defined.

The following output illustrates such a coupling.[36]

```
**SETQ(ALTFORM (MAPCONC NASDIC (F/L (GETP X 'ALTFORMS]      [1]
=NASDICT
(AL26 BE7 CO56 CO57 CO60 C13 H3 MN54 NA22 SC46 S34 TI44)
**(GIVEMELINESCONTAININGCOBALT)                              [2]
SAMPLE   PHASE    CONSTIT.  CONTENT  UNIT    CITATION  TAG
S10002   OVERALL  CO56      40.0     DPM/KG  D70-237   0
                  C13       8.8      DEL     D70-228   0
                  H3        314.0    DPM/KG
                  MN54      28

**GETP(COBALT ALTFORMS)                                      [3]
(CO56 CO57 CO60 C13 H3 MN54 NA22 SC46 S34 T144)
**UNDO MAPCONC                                               [4]
SETQ UNDONE.
**REDOGETP                                                   [5]
(CO56 CO57 CO60)
**REDO COBALT                                                [6]
SAMPLE   PHASE    CONSTIT.  CONTENT  UNIT    CITATION  TAG
S10002   OVERALL  CO57      40.0     DPM/KG  D70-237   0
S10003   OVERALL  CO        15.0             D70-203   0
                            14.1             D70-216
                  CO56      43.0     DPM/KG  D70-237   0
                  CO57      43.0             D70-241   0
                  CO60      1.0
**USE MANGANESE FOR COBALT
```

The user is running under his own executive program which accepts requests in the form of sentences, which it first parses and then executes. The user first 'innocently' computes a list of all ALTERNATIVE-FORMS for the elements in his system [1]. He then inputs a request in sentence format [2] expecting to see under the column CONSTIT. only cobalt, CO, or its alternate forms, CO56, CO57, or CO60. Seeing C13, H3, and MN54, he aborts the output, and checks the property ALTFORMS for COBALT [3]. The appearance of C13, H3, MN54, he aborts the output, and checks the property ALTFORMS for COBALT [3]. The appearance of C13, H3, MN54 et al, remind him that the mapconc is destructive, and that in the process of making a list of the ALTFORMS, he has inadvertently strung them all together. Recovering from this situation would require him to individually

---

[36] The output is from the Lunar Sciences Natural Language Information System being developed for the NASA Manned Spacecraft Center by William A. Woods of Bolt Beranek and Newman Inc., Cambridge, Mass.

examine and correct the ALTFORMs for each element in his dictionary, a tedious process. Instead, he can simply UNDO MAPCONC, [4] check to make sure the ALTFORM has been corrected [5], then redo his original request [6] and continue. The UNDO is possible because the first input was executed by lispx; the (GIVE ME LINES CONTAINING COBALT) is possible because the user defined lispxuserfn appropriately; and the REDO and USE are possible because the (GIVE ME LINES CONTAINING COBALT) was stored on the history list before it was transmitted to lispxuserfn and the user's parsing program.

lispxuserfn is a function of two arguments, x and line, where x is the first expression typed, and line the rest of the line, as read by readline (see page 22.47). For example, if the user types FOO(A B C), x=FOO, and line=((A B C)); if the user types (FOO A B C), x=(FOO A B C), and line=NIL; and if the user types FOO A B C, x=FOO and line=(A B C).

Thus in the above example, lispxuserfn would be defined as:

```
[LAMBDA (X LINE)
    (COND
      ((AND (NULL LINE)
            (LISTP X))
          (SETQ LISPXVALUE (PARSE X))
         T]
```

Note that since lispxuserfn is called for each input (except for p.a. commands), it can also be used to monitor some condition or gather statistics.

* * *

In addition to saving inputs and values, lispx saves most system messages on the history list, e.g. FILE CREATED --, (fn REDEFINED), (var RESET), output of TIME, BREAKDOWN, STORAGE, DWIM messages, etc. When printhistory prints the event, this output is replicated. This facility is implemented via the functions lispxprint, lispxprin1, lispxprin2, lispxspaces, lispxterpri, and

lispxtab.[37] In addition to performing the corresponding output operation, these functions store an appropriate expression on the history event under the property LISPXPRINT.[38] This expression is used by printhistory to reproduce the output.

* * *

In addition to the above features, lispx checks to see if car or cdr of NIL or car of T have been clobbered, and if so, restores them and prints a message. Lispx also performs spelling corrections using lispxcoms, a list of its commands, as a spelling list whenever it is given an unbound atom or undefined function, i.e. before attempting to evaluate the input.[39]

22.6 Undoing

The UNDO capability of the programmer's assistant is implemented by requiring that each operation that is to be undoable be responsible itself for saving on the history list enough information to enable reversal of its side effects. In other words, the assistant does not 'know' when it is about to perform a destructive operation, i.e. it is not constantly checking or anticipating. Instead, it simply executes operations, and any undoable changes that occur are

---

[37] In fact, all six of these functions have the same definition. When called, this function looks back on the stack to see what name it was called by, and determines what to do. Thus, if the user wanted to make any other output function, e.g. printdef, record its MOVD(LISPXPRINT LISPXPRINTDEF), and then use lispxprintdef for printdef. (This will work only for functions of three or fewer arguments.)

[38] unless lispxprintflg is NIL.

[39] lispx is also responsible for rebinding helpclock, used by breakcheck, Section 16, for computing the amount of time spent in a computation, in order to determine whether to go into a break if and when an error occurs.

automatically saved on the history list by the responsible function.[40] The operation of UNDOing, which involves recovering the saved information and performing the corresponding inverses, works the same way, so that the user can UNDO an UNDO, and UNDO that etc.

At each point, until the user specifically requests an operation to be undone, the assistant does not know, or care, whether information has been saved to enable the undoing. Only when the user attempts to undo an operation does the assistant check to see whether any information has been saved. If none has been saved, and the user has specifically named the event he wants undone, the assistant types NOTHING SAVED. (When the user simply types UNDO, the assistant searches for the last undoable event, ingnoring events already undone as well as UNDO operations themselves.)

This implementation minimizes the overhead for undoing. Only those operations which actually make changes are affected, and the overhead is small: two or three cells of storage for saving the information, and an extra function call. However, even this small price may be too expensive if the operation is sufficiently primitive and repetitive, i.e. if the extra overhead may seriously degrade the overall performance of the program.[41] Hence not every destructive operation in a program should necessarily be undoable; the programmer must be allowed to decide each case individually.

---

[40] When the number of changes that have been saved exceeds the value of #undosaves (initially set to 50), the user is asked if he wants to continue saving the undo information for this event. The purpose of this feature is to avoid tying up large quantities of storage for operations that will never need to be undone. The interaction is handled by the same routines used by DWIM, so that the input buffers are first saved and cleared, the message typed, then the system waits dwimwait seconds, and if there is no response, assumes the default answer, which in this case is NO. Finally the input buffers are restored. See page 22.56 for details.

[41] The rest of the discussion applies only to lispx; the editor handles undoing itself in a slightly different fashion, as described on page 22.61.

Therefore for each primitive destructive operation, we have implemented *two* separate functions, one which always saves information, i.e. is always undoable, and one which does not, e.g. /rplaca and rplaca, /put and put.[42] In the various system packages, the appropriate function is used. For example, break uses /putd and /remprop so as to be undoable, and DWIM uses /rplaca and /rplacd, when it makes a correction.[43] Similarly the user can simply use the corresponding / function if he wants to make a destructive operation in his own program undoable. When the / function is called, it will save the undo information in the current event on the history list.

However, all operations that are *typed in* to lispx *are* made undoable, simply by substituting the corresponding / function [44] for any destructive function throughout the input.[45] For example, on page 22.8, when the user typed (MAPCONC NASDIC (F/L ...)) it was (/MAPCONC NASDIC (F/L ...)) that was evaluated. Since the system cannot know whether efficiency and overhead are serious considerations for the execution of an expression in a user *program*, the user must decide, e.g. call /mapconc if he wants the operation undoable.

---

42 The 'slash' functions currently implemented are /addprop, /attach, /dremove, /dreverse, /dsubst, /lconc, /mapcon, /mapconc, /movd, /nconc, /nconc1, /put, /putd, /putdq, /puthash, /remprop, /rplaca, /rplacd, /set, /seta, /setd, and /tconc. Note that /setq and /setqq are *not* included. If the user wants a set operation undoable in his program, he must see /set, or /rplaca.

43 The effects of the following functions are always undoable (regardless of whether or not they are typed in): define, defineq, defc (used to give a function a compiled code definition), deflist, load, savedef, unsavedef, break, unbreak, rebreak, trace, breakin, unbreakin, changename, editfns, editf, editv, editp, edite, editl, esubst, advise, unadvise, readvise, plus any changes caused by DWIM.

44 Since there is no /setq, setq's appearing in type-in are handled specially by substituting a call to saveset, page 22.43.

45 The substitution is performed by the function lispx/, described on page 22.58.

However, expressions that are typed-in rarely involve iterations or lengthy computations *directly*. Therefore, if all primitive destructive functions that are immediately contained in a type-in are made undoable, there will rarely be a significant loss of efficiency. Thus lispx scans all user input before evaluating it, and substitutes the corresponding undoable function for all primitive destructive functions. Obviously with a more sophisticated analysis of both user input and user programs, the decision concerning which operations to make undoable could be better advised. However, we have found the configuration described here to be a very satisfactory one. The user pays a very small price for being able to undo what he types in, and if he wishes to protect himself from malfunctioning in his own programs, he can have his program specifically call undoable functions, or go into testmode as described next.

Testmode

Because of efficiency considerations, the user may not want certain functions undoable after his program becomes operational. However, while debugging he may find it desirable to protect himself against a program running wild, by making primitive destructive operations undoable. The function testmode provides this capability by temporarily making everything undoable.

testmode[flg] — testmode[T] redefines all primitive destructive functions[46] with their corresponding undoable versions and sets testmodeflg to T. testmode[] restores the original definitions, and sets testmodeflg to NIL.[47]

---

46 i.e. the 'slash' functions; see footnote on page 22.40.

47 testmode will have no effect on compiled mapconc's, since they compile open with frplacd's.

Note that setq's are *not* undoable, even in testmode. To make the corresponding operation undoable in testmode, set or rplaca should be used.

Undoing Out of Order

/rplaca and /rplacd operate by saving the pointer that is to be changed and its original contents (i.e. /rplaca saves car and /rplacd saves cdr). Undoing /rplaca and /rplacd simply restores the pointer. Thus, if the user types (RPLACA FOO 1), followed by (RPLACA FOO 2), then undoes both events by undoing the most recent event first, then undoing the older event, FOO will be restored to its state before either rplaca operated. However if the user undoes the first event, *then* the second event, (CAR FOO) will be 1, since this is what was in car of FOO before (RPLACA FOO 2) was executed. Similarly, if the user performs (NCONC1 FOO 1) then (NCONC1 FOO 2), undoing just (NCONC1 FOO 1) will remove both 1 and 2 from FOO. The problem in both cases is that the two operations are not 'independent.' In general, operations are always independent if they affect different lists or different sublists of the same list.[48] Undoing in reverse order of execution, or undoing independent operations, is always guaranteed to do the 'right' thing. However, undoing dependent operations out of order may not always have the predicted effect.

---

[48] Property list operations, (i.e. put, addprop and remprop) are handled specially so that they are always independent, even when they affect the same property list. For example, if the user types PUT(FOO FIE1 FUM1) then PUT(FOO FIE2 FUM2), then undoes the first event, the FIE2 property will remain, even though CDR(FOO) may have been NIL at the time the first event was executed.

Saveset

Setq's are made undoable on type in by substituting a call to saveset (described in detail on page 22.55), whenever setq is the name of the function to be applied, or car of the form to be evaluated. In addition to saving enough information on the history list to enable undoing, saveset operates in a manner analogous to savedef when it resets a top level value, i.e. when it changes a top level binding from a value other than NOBIND to a new value that is not equal to the old one. In this case, saveset saves the old value of the variable being set on the variable's property list under the property VALUE, and prints the message (variable RESET). The old value can be restored via the function unset,[49] which also saves the current value (but does not print a message). Thus unset can be used to flip back and forth between two values.

rpaq and rpaqq are implemented via calls to saveset. Thus old values will be saved and messages printed for any variables that are reset as the result of loading a file.[50] Calls to set and setqq appearing in type in are also converted to appropriate calls to saveset.

For top level variables, saveset also adds the variable to the appropriate spelling list, thereby noticing variables set in files via rpaqq or rpaqq, as well as those set via type in.

---

49 Of course, UNDO can be used as long as the event containing this call to saveset is still active. Note however that the old value will remain on the property list, and therefore be recoverable via unset, even after the original event has been forgotten.

50 To complete the analogy with define, saveset will not save old values on property lists if dfnflg=T, e.g. when load is called with second argument T, (however, the call to saveset will still be undoable,) and when dfnflg=ALLPROP, the value is stored directly on the property list under property VALUE (the latter applies only to calls from rpaqq and rpaq).

22.7 Format and Use of the History List

There are currently two history lists, lispxhistory and edithistory. Both history lists have the same format, and in fact, each use the same function, historysave, for recording events, and the same set of functions for implementing commands that refer to the history list, e.g. historyfind, printhistory, undosave, etc.[51]

Each history list is a list of the form (l event# size mod), where l is the list of events with the most recent event first, event# is the event number for the most recent event on l, size is the size of the time-slice, i.e. the maximum length of l, and mod is the highest possible event number (see footnote on page 22.8). lispxhistory and edithistory are both initialized to (NIL 0 30 100). Setting lispxhistory or edithistory to NIL is permitted, and simply disables all history features, i.e. lispxhistory and edithistory act like flags as well as repositories of events.

Each individual event on l is a list of the form (input id value . props), where input is the input sequence for the event, as described on page 22.17-19, id the prompt character, e.g. ←, :, *,[52] and value is the value of the event, and is initialized to bell.[53]

---

[51] A third history list, archivelst, is used when events are archived, as described on page 22.27. It too uses the same format.

[52] id is one of the arguments to lispx and to historysave. A user can call lispx giving it any prompt character he wishes (except for *, since in certain cases, lispx must use the value of id to tell whether or not it was called from the editor.) For example, on page 22.36, the user's prompt character was **.

[53] On edithistory, this field is used to save the side effects of each command.

props is a property list, i.e. of the form (property value property value --). props can be used to associate arbitrary information with a particular event. Currently, the properties SIDE, GROUP, HISTORY, PRINT, USE-ARGS, ...ARGS, ERROR, and LISPXPRINT are being used. The value of property SIDE is a list of the side effects of the event. (See discussion of undosave, page 22.56, and undolispx, page 22.58). The HISTORY and GROUP properties are used for commands that reexecute previous events, i.e. REDO, RETRY, USE, ..., and FIX. The value of the HISTORY property is the history command itself, i.e. what the user actually typed, e.g. REDO FROM F, and is used by the ?? command for printing the event. The value of the property PRINT is also for use by the ?? command, when special formatting is required, for example, in printing events corresponding to the break commands OK, GO, EVAL, and ?=. USE-ARGS and ...ARGS are used to save the arguments and expression for the corresponding history command. ERROR is used by the $ command. LISPXPRINT is used to record calls to lispxprint, lispxprin1, et al, See page 22.37.

When lispx is given an input, it calls historysave to record the input in a new event.[54] Normally, historysave returns as its value cddr of the new event, i.e. car of its value is the value field of the event. lispx binds lispxhist to the value of historysave, so that when the operation has completed, lispx knows where to store the value, namely in car of lispxhist.[55] lispxhist also provides access to the property list for the current event. For example, the / functions are all implemented to call undosave, which simply adds the corresponding information to lispxhist under the property SIDE, or if there is no property SIDE, creates one, and then adds the information.

---

[54] The commands ??, FORGET, TYPE-AHEAD, $BUFS, and ARCHIVE are executed immediately, and are not recorded on the history list.

[55] Note that by the time it completes, the operation may no longer correspond to the most recent event on the history list. For example, all inputs typed to a lower break will appear later on the history list.

After binding lispxhist, lispx executes the input, stores its value in car of lispxhist, prints the value, and returns.

When the input is a REDO, RETRY, USE, ..., or FIX command, the procedure is similar, except that the event is also given a GROUP property, initially NIL, and a HISTORY property, and lispx simply unreads the input and returns. When the input is 'reread', it is historysave, not lispx, that notices this fact, and finds the event from which the input originally came.[56] historysave then adds a new (value . props) entry to the GROUP property for this event, and returns this entry as the 'new event.' lispx then proceeds exactly as when its input was typed directly, i.e. it binds lispxhist to the value of historysave, executes the input, stores the value in car of lispxhist, prints the value, and returns. In fact, lispx never notices whether it is working on freshly typed input, or input that was reread. Similarly, undosave will store undo information on lispxhist under the property SIDE the same as always, and does not know or care that lispxhist is not the entire event, but one of the elements of the GROUP property. Thus when the event is finished, its entry will look like:

```
(input id value HISTORY command GROUP ((value1 SIDE side1)
                                       (value2 SIDE side2)
                                           ...))              57
```

This implementation removes the burden from the function calling historysave of distinguishing between new input and reexecution of input whose history entry

---

[56] If historysave cannot find the event, for example if a user program unreads the input directly, and not via a history command, historysave proceeds as though the input were typed.

[57] In this case, the value field is not being used; valueof instead collects each of the values from the GROUP property, i.e. returns mapcar[get[event;GROUP];CAR]. Similarly, undo operates by collecting the SIDE properties from each of the elements of the GROUP property, and then undoing them in reverse order.

has already been set up.[58]

22.8 lispx and readline

lispx is called with the first expression typed on a line as its first argument, lispxx.

If this is *not* a list, lispx *always* does a readline, and treats lispxx plus the line as the input for the event, and stores it accordingly on the history list.[59] *Then* it decides what to do with the input, i.e. if it is not recognized as a command, a lispxmacro, or is processed by lispxuserfn, call eval or apply.[60] readline normally is terminated either by (1) a carriage return that is not preceeded by a space, or (2) a list that is terminated by a], or (3) an unmatched) or], which is not included in the line. However, when called from lispx, readline operates differently in two respects:

(1) If the line consists of a single) or], readline returns (NIL) instead of NIL, i.e. the) or] *is* included in the line. This permits the user to type FOO) or FOO], meaning call the function' FOO with no arguments, as opposed to FOO↩ (FOOcarriage-return), meaning evaluate the variable FOO.

(2) If the first expression on the line is a list that is not preceded by

---

[58] Although we have not yet done so, this implementation, i.e. keeping the various 'sub-events' separate with respect to values and properties, also permits constructing commands for operating on just one of the sub-events.

[59] If lispxx is a list car of which is LAMBDA or NLAMBDA, lispx calls lispxread to obtain the arguments.

[60] If the input consists of one expression, eval is called; if two, apply; if more than two, the entire line is treated as a single form and eval is called.

any spaces, the list terminates the line regardless of whether or not it is terminated by]. This permits the user to type EDITF(FOO) as a single input.

Note that if any spaces are inserted between the atom and the left parentheses or bracket, readline will assume that the list does not terminate the line. This is to enable the user to type a line command such as USE (FOO) FOR FOO. In this case, a carriage return will be typed after (FOO) followed by "..." as described in Section 14. Therefore, if the user accidentially puts an extra space between a function and its arguments, he will have to complete the input with another carriage return, e.g.

```
←EDITF_(FOO)
...)
EDIT
*
```

22.9 Functions

lispx[lispxx;lispxid;lispxxmacros;lispxxuserfn][61]

lispx is like eval/apply. It carries out a single computation, and returns its value. The first argument, lispxx is the result of a single call to lispxread. lispx will call readline, if necessary as described on page 22.47. lispx prints the value of the computation, as well as saving the

---

[61] lispxid corresponds to id on page 22.44. Lispx also has a fifth argument, lispxflg, which is used by the E command in the editor.

input and value on lispxhistory.[62]

If lispxx is a history command, lispx executes the command, and returns bell as its value.
If the value of the fourth argument, lispxxmacros, is not NIL, it is used as the lispx macros, otherwise the top level value of lispxmacros is used. If the value of the fifth argument, lispxxuserfn, is not NIL, it is used as lispxuserfn. In this case, it is not necessary to both set and define lispxuserfn as described on page 22.35.

The overhead for a call to lispx is approximately 17 milliseconds, of which 12 milliseconds are spent in maintaining the spelling lists. In other words, in INTERLISP, the user pays 17 more milliseconds for each eval or apply input over a conventional LISP executive, in order to enable the features described in this chapter.

userexec[lispxid;lispxxmacros;lispxxuserfn]

repeatedly calls lispx under errorset protection specifying lispxxmacros and lispxxuserfn, and using lispxid (or ← if lispxid=NIL) as a prompt character. Userexec is exited via the lispxmacro OK, or else with a retfrom.

---

[62] Note that the history is *not* one of the arguments to lispx, i.e. the editor must bind (reset) lispxhistory to edithistory before calling lispx to carry out a history command.
Lispx will continue to operate as an eval/apply function if lispxhistory is NIL. Only those functions and commands that involve the history list will be affected.

lispxread[file] is a generalized read. If readbuf=NIL, lispxread performs read[file], which it returns as its value. (If the user types control-U during the call to read, lispxread calls the editor and returns the edited value.)

If readbuf is not NIL, lispxread 'reads' the next expression on readbuf, i.e. essentially returns

```
(PROG1 (CAR READBUF)
       (SETQ READBUF (CDR READBUF))).
```
[63]

readline, described in Section 14, also uses this generalized notion of reading. When readbuf is not NIL, readline 'reads' expressions from readbuf until it either reaches the end of readbuf, or until it reads a (VAG 0). In both cases, it returns a list of the expressions it has 'read'. (The (VAG 0) is not included in the list.)

When readbuf is not NIL, both lispxread and readline actually obtain their input by performing (APPLY* LISPXREADFN FILE), where lispxreadfn is initially set to READ. Thus, if the user wants lispx, the editor, break, et al to do their reading via a different input function, e.g. uread, he simply sets lispxreadfn to the name of that function (or an appropriate LAMBDA expression).

lispxreadp[flg] is a generalized readp. If flg=T, lispxreadp returns T if there is any input waiting to be 'read', a la lispxread. If flg=NIL, lispxreadp returns T only if there is any input waiting to be 'read' *on this line*. In both cases, leading spaces

---

[63] Except that pseudo-carriage returns, as represented by (VAG 0), are ignored, i.e. skipped. Lispxread also sets rereadflg to NIL when it reads via read, and sets rereadflg to the value of readbuf when rereading.

are ignored, i.e. skipped over with readc, so that if only spaces have been typed, lispxreadp will return NIL.

lispxunread[lst] — unreads lst, a list of expressions to be read. If readbuf is not NIL, lispxunread attaches lst at the front of readbuf, separating it from the rest of readbuf with a (VAG 0). The definition of lispxunread is:

```
(LISPXUNREAD
  [LAMBDA (LST)
    (SETQ READBUF (COND
        ((NULL READBUF)
          LST)
        (T (APPEND LST (CONS (VAG 0)
                              READBUF])
```

promptchar[id;flg;hist] — prints the prompt character id.

promptchar will not print anything when the next input will be 'reread', i.e. readbuf is not NIL. promptchar will also not print when readp[ ]=T, unless flg is T.

Thus the editor calls promptchar with flg=NIL so that extra *'s are not printed when the user types several commands on one line. However, evalqt calls promptchar with flg=T since it always wants the ← printed (except when 'rereading').

Finally, if prompt#flg is T and hist is not NIL, promptchar prints the current event number (of hist) before printing id.

lispxeval[lispxform;lispxid] evaluates lispxform (using eval) the same as though it were typed in to lispx, i.e. the event is recorded, and the evaluation is made undoable by substituting the slash functions for the corresponding destructive functions, as described on page 22.40. lispxeval returns the value of the form, but does not print it.

historysave[history;id;input1;input2;input3;props]

records one event on history. If input1 is not NIL, the input is of the form (input1 input2 . input3). If input1 is NIL, and input2 is not NIL, the input is of the form (input2 . input3). Otherwise, the input is just input3.

historysave creates a new event with the corresponding input, id, value field initialized to bell, and props. If the history has reached its full size, the last event is removed and cannibalized.

The value of historysave is cddr of the event. However, if rereadflg is not NIL, and is a tail of the input of the most recent event on the history list, and this event contains a GROUP property, historysave does not create a new event, but simply adds a (bell . props) entry to the GROUP property and returns that entry. See discussion on page 22.46.

lispxfind[history;line;type;backup]

line is an event specification, type specifies the format of the value to be returned by lispxfind, and can be either ENTRY, ENTRIES, COPY, COPIES, INPUT, or REDO. lispxfind parses line, and uses historyfind to find the corresponding events. lispxfind then assembles and returns the appropriate structure.

lispxfind incorporates the following special features:

(1) if backup=T, lispxfind interprets line in the context of the history list *before* the current event was added. This feature is used, for example, by valueof, so that (VALUEOF -1) will not refer to the valueof event itself;

(2) if line=NIL and the last event is an UNDO, the next to the last event is taken. This permits the user to type UNDO followed by REDO or USE;

(3) lispxfind recognizes @@, and substitutes archivelst for history (see page 22.14); and

(4) lispxfind recognizes @, and retrieves the corresponding event(s) from the property list of the atom following @ (see page 22.14).

historyfind[lst;index;mod;x;y]

searches lst and returns the tails of lst beginning with the event corresponding to x. lst, index, and mod are as described on page 22.44. x is an event address, as described on page 22.11-14, e.g. (43), (-1), (FOO FIE),

(LOAD ← FOO), etc.[64] If historyfind cannot find x, it generates an error.

entry#[hist;x] — hist is a history list, i.e. of the form described on page 22.44. x is one of the events on hist, i.e. (MEMB X (CAR HIST)) is true. The value of entry# is the event number for x.

valueof[x] — is an nlambda, nospread function for obtaining the value of the event specified by x, e.g. (VALUEOF -1), (VALUEOF LOAD 1), etc. valueof returns a list of the corresponding values if x specifies a multiple event.

changeslice[n;history][65] — changes time-slice for history to n. If history is NIL, changes both edithistory and lispxhistory.

Note: the effect of *increasing* a time-slice is gradual: the history list is simply allowed to grow to the corresponding length before any events are forgotten. *Decreasing* a time-slice will immediately remove a sufficient number of the older events to bring the history list down to the proper size. However, changeslice is undoable, so that these events are (temporarily) recoverable. Thus if the user wants to recover the storage associated with these events without waiting n more events for the changeslice event to be forgotten, he must perform a FORGET command.

---

[64] If y is given, the event address is the *list difference* between x and y, e.g. x=(FOO FIE AND \ -1), y=(AND \ -1) is equivalent to x=(FOO FIE), y=NIL.

[65] changeslice has a third argument used by the system for undoing a changeslice.

saveset[name;value;topflg;flg]

an undoable set. (see page 22.43). saveset scans the pushdown list looking for the last binding of name, sets name to value, and returns value.

If the binding changed was a top level binding, name is added to spellings3 (see Section 17). Furthermore, if the old value was not NOBIND, and was also not equal to the new value, saveset calls the file package to update the necessary file records. Then, if dfnflg is not equal to T, saveset prints (name RESET), and saves the old value on the property list of name, under the property VALUE. If flg=NOPRINT, saveset saves the old value, but does not print the message. This option is used by unset.

If topflg=T, saveset operates as above except that it does not scan the pushdown list but goes right to name's value cell, e.g. rpaqq[x;y] is simply saveset[x;y;T]. When topflg is T, and dfnflg is ALLPROP and the old value was not NOBIND, saveset simply stores value on the property list of name under the property VALUE, and returns value. This option is used for loading files without disturbing the current value of variables (see Section 14).

If flg=NOSAVE, saveset does *not* save the old value on the property list, nor does it add name to spellings3. However, the call to saveset is still undoable. This option is used by /set.

unset[name] — if name does not contain a property VALUE, unset generates an error. Otherwise unset calls saveset with name, the property value, topflg=T, and flg=NOPRINT.

undosave[undoform][66] — if lispxhist is not NIL (see discussion on page 22.45), and get[lispxhist;SIDE] is not equal to NOSAVE, undosave adds undoform to the value of the property SIDE on lispxhist, creating a SIDE property if one does not already exist. The form of undoform is (fn . args),[67] i.e. undoform is undone by performing apply[car[undoform];cdr[undoform]]. For example, if the definition of FOO is def, /putd[FOO;newdef] will cause a call undosave with undoform =(/PUTD FOO def).

car of the SIDE property is the number of 'undosaves', i.e. length of cdr of the SIDE property, which is the list of undoforms. Each call to undosave increments this count, and adds undoform to the front of the list, i.e. just after the count. When the count reaches the value of #undosaves (initially 50),[68] undosave prints a

---

66 Undosave has a second optional argument, histentry, which can be used to specify lispxhist directly, saving the time to look it up. If both histentry and lispxhist are NIL, undosave is a NOP.

67 Except for /rplnode, as described below.

68 #undosaves=NIL is equivalent to #undosaves=infinity.

message asking the user if he wants to continue saving. If the user answers NO or defaults, undosave makes NOSAVE be the value of the property SIDE, which disables any further saving for this event. If the user answers YES, undosave changes the count to -1, which is then never incremented, and continues saving.[69]

/rplnode[x;a;d]

Undoably performs rplaca[x;a] and rplacd[x;d]. Value is x. The principle advantage of /rplnode is that when x is a list, /rplnode saves its undo information as cons[x;cons[car[x];cdr[x]]], i.e. (x originalcar . originalcdr), and therefore requires only 3 cells of storage, instead of the 8 that would be required for a /rplaca and a /rplacd that saved their information as described earlier.[70]

/rplnode has a BLKLIBRARYDEF.

new/fn[fn]

After the user has defined /fn, new/fn performs the necessary housekeeping operations to make fn be undoable.

---

69 load initializes the count on SIDE to -1, so that regardless of the value of #undosaves, no message will be printed, and the load will be undoable.

70 Actually, /rplaca and /rplacd also use this format for saving their undo information when their first arguments are lists. However, if both a /rplaca and /rplacd are to be performed, it is still more efficient to use /rplnode (3 cells versus 6 cells).

For example, the user could define /radix as (LAMBDA (X) (UNDOSAVE (LIST (QUOTE /RADIX) (RADIX X))) and then perform new/fn[radix], and radix would then be undoable when typed in or in testmode.

lispx/[x;fn;vars]

performs the substitution of / functions for destructive functions. If fn is not NIL, it is the name of a function, and x is its argument list. If fn is NIL, x is a form. In both cases, lispx/ returns x with the appropriate substitutions. Vars is a list of bound variables (optional).

lispx/ incorporates information about the syntax and semantics of INTERLISP expressions. For example, it does not bother to make undoable operations involving variables bound in x. It also knows that substitution should not be performed inside of expressions car of which is QUOTE, DEFINEQ, BREAK, etc.[71] Similarly, substitution *should* be performed in the arguments for functions like mapc, rptq, etc., since these contain expressions that will be evaluated or applied. For example, if the user types mapc[(FOO1 FOO2 FOO3);PUTD] the putd must be replaced by /putd.

undolispx[line]

line is an event specification. undolispx is the function that executes UNDO commands by calling undolispx1 on the appropriate entry(s).

---

[71] Any member of the list nosubstfns.

undolispx1[event;flg]

undoes one event. The value of undolispx1 is NIL if there is nothing to be undone. If the event is already undone, undolispx1 prints ALREADY UNDONE and returns T.[72] Otherwise, undolispx1 undoes the event, prints a message, e.g. SETQ UNDONE, and returns T.

Undoing an event consists of mapping down (cdr of) the property value for SIDE, and for each element, applying car to cdr, and then marking the event undone by attaching (with /attach) a NIL to the front of its SIDE property. Note that the undoing of each element on the SIDE property will usually cause undosaves to be added to the *current* lispxhist, thereby enabling the effects of undolispx1 to be undone.

undonlsetq[form]

is an nlambda function similar to nlsetq. undonlsetq evaluates form, and if no error occurs during the evaluation, returns list[eval[form]] and passes the undo information from form (if any) upwards.[73] If an error does occur, the value of undonlsetq is NIL, and any changes made by / functions during the evaluation of form are undone.

---

[72] If flg=T and the event is already undone, or is an undo command, undolispx1 takes no action and returns NIL. Undolispx uses this option to search for the last event to undo. Thus when line=NIL, undolispx simply searches history until it finds an event for which undolispx1 returns T, i.e. undolispx performs (SOME (CDAR LISPXHISTORY) (F/L (UNDOLISPX1 X T)))

[73] Actually, undonlsetq does not rebind lispxhist, so that any undo information is stored directly on the history event, exactly as though there were no undonlsetq. Instead, undonlsetq simply marks the state of the undo information when it starts, so that if an error occurs, it can then know how much to undo. The purpose of this is so that if the user control-D's out of the undonlsetq, the event is still undoable.

undonlsetq compiles open.

undonlsetq will operate even if lispxhistory or lispxhist are NIL, or if #undosaves is or has been exceeded for this event.

Note that undonlsetq provides a limited form of backtracking.

printhistory[history;line;skipfn;novalues]

line is an event specification. printhistory prints the events on history specified by line, e.g. (-1 THRU -10). skipfn is an (optional) functional argument that is applied to each event before printing. If its value is true, the event is skipped, i.e. not printed. If novalues=T, or novalues applied to the corresponding event is true, the value is not printed.[74]

For example, the following lispxmacro will define ??' as a command for printing the history list while skipping all 'large events' and not printing any values.

```
(??' (PRINTHISTORY LISPXHISTORY LISPXLINE
       (FUNCTION (LAMBDA (X)
         (IGREATERP (COUNT (CAR X)) 5)))
      T))
```

---

[74] For example, novalues is T when printing events on edithistory.

22.10 The Editor and the Assistant

As mentioned earlier, all of the remarks concerning 'the assistant' apply equally well to user interactions with evalqt, break or the editor. The differences between the editor's implementation of these features and that of lispx are mostly obvious or inconsequential. However, for completeness, this section discusses the editor's implementation of the programmer's assistant.

The editor uses promptchar to print its prompt character, and lispxread, lispxreadp, and readline for obtaining inputs. When the editor is given an input, it calls historysave to record the input in a new event on its history list, edithistory.[75] Edithistory follows the same conventions and format as lispxhistory. However, since edit commands have no value, the editor uses the value field for saving side effects, rather than storing them under the property SIDE.

The editor processes DO, !E, !F, and !N commands itself, since lispx does not recognize these commands. The editor also processes UNDO itself, as described below. All other history commands[76] are simply given to lispx for execution, after first binding (resetting) lispxhistory to edithistory. The editor also

---

[75] Except that the atomic commands OK, STOP, SAVE, P, ?, PP and E are not recorded. In addition, number commands are grouped together in a single event. For example, 3 3 -1 is considered as one command for changing position.

[76] as indicated by their appearance on historycoms, a list of the history commands. editdefault interrogates historycoms before attempting spelling correction. (All of the commands on historycoms are also on editcomsa and editcomsl so that they can be corrected if misspelled in the editor.) Thus if the user defines a lispxmacro and wishes it to operate in the editor as well, he need simply add it to historycoms. For example, RETRIEVE is implemented as a lispxmacro and works equally well in lispx and the editor.

calls lispx when given an E command as described in Section 9.[77]

The major implementation difference between the editor and lispx occurs in undoing. Edithistory is a list of only the last n commands, where n is the value of the time-slice. However the editor provides for undoing *all* changes made in a single editing session, even if that session consisted of more than n edit commands. Therefore, the editor saves undo information independently of the edithistory on a list call undolst, (although it also stores each entry on undolst in the field of the corresponding event on edithistory.) Thus, the commands UNDO, !UNDO, and UNBLOCK, are not dependent on edithistory,[78] i.e. UNDO specifies undoing the last command on undolst, even if that event no longer appears on edithistory. The only interaction between UNDO and the history list occurs when the user types UNDO followed by an event specification. In this case, the editor calls lispxfind to find the event, and then undoes the corresponding entry on undolst. Thus the user can only undo a *specified* command within the scope of the edithistory. (Note that this is also the only way UNDO commands themselves can be undone, that is, by using the history feature, to specify the corresponding event, e.g. UNDO UNDO.)

The implementation of the actual undoing is similar to the way it is done in lispx: each command that makes a change in the structure being edited does so via a function that records the change on a variable. After the command has completed, this variable contains a list of all the pointers that have been

---

[77] In this case, the editor uses the fifth argument to lispx, lispxflg, to specify that any history commands are to be executed by a recursive call to lispx, rather than by unreading. For example, if the user types E REDO in the editor, he wants the last event on lispxhistory processed as lispx input, and not to be unread and processed by the editor.

[78] and in fact will work if edithistory=NIL, or even in a system which does not contain lispx at all.

changed and their original contents. Undoing that command simply involves mapping down that list and restoring the pointers.

22.11 Statistics

The programmer's assistant keeps various statistics about system usage, e.g. number of lispx inputs, number of undo saves, number of calls to editor, number of edit commands, number of p.a. commands, cpu time, console time, et al. These can be viewed via the function lispxstats.

lispxstats[] — prints statistics.

The user can add his own statistics to the lispx statistics via the function addstats.

addstats[statlst] — no spread, nlambda. Statlst is a list of elements of the form (statistic-name . message), e.g. (EDITCALLS CALLS TO EDITOR) (UNDOSTATS CHANGES UNDONE), etc. statistic-name is set to the cell in an unboxed array, where the corresponding statistic will be stored. This statistic can then be incremented by lispxwatch.

lispxwatch[stat;n] — increments stat by n (or 1 if n=NIL). lispxwatch has a BLKLIBRARYDEF.

The user can save his statistics for loading into a new system by performing MAKEFILE(DUMPSTATS). After the file DUMPSTATS is loaded, the statistics printed by lispxstats will be the same as those that would be printed following the makefile.

22.12 Greeting and User Initialization

Many of the features of INTERLISP are parameterized to allow the user to adjust the system to his or her own tastes. Among the more commonly adjusted parameters are prompt#flg, dwimwait, changeslice, #rpars, lowercase, archivefn, #undosaves, fltfmt, etc. In addition, the user can modify the action of system functions in ways not specifically provided for by using advise (Section 19).

In order to encourage this procedure, and to make it as painless and automatic as possible, the p.a. includes a facility for a user-defined profile. When INTERLISP is first run, a specially formatted file on the LISP file directory is indexed into using the user's usernumber as a key. The expressions (if any) found there are then evaluated, and the p.a. prints a greeting, e.g., "HELLO, WARREN." or "GOOD AFTERNOON, DANNY.", etc.

Greeting (i.e., the initialization) is undoable, and is stored as a separate event on the history list. The user can also specifically invoke the greeting operation via the function greet, for example, if he wishes to effect another user's initialization.

greet[name;flg] — performs greeting for user whose username is name, or if name=NIL, for login name (see username and usernumber, Section 21), i.e., when INTERLISP first starts up, it performs greet[]. Before greet performs the indicated initialization, it first undoes the effects of the previous greeting.[79] If flg=T, greet also resets the counters for the various statistics reported by lispxstats (page 22.63).

---

[79] The side effects of the greeting operation are stored on a global variable as well as the history list, thus enabling the previous greeting to be undone even if it is no longer on the history list.

greet also sets the variable username to the name for which the greeting was performed. Sysin is advised to compare username with username[], and to print a message alerting the user if they are not the same. For example, if user HARTLEY performs a sysin of a sysout made by user GOODWIN, the following message is printed:

```
****ATTENTION USER HARTLEY:
THIS SYSOUT IS INITIALIZED FOR USER GOODWIN.
TOREINITIALIZE,TYPEGREET()
```
[80]

Implementation

greet operates off the file <LISP>USERNAMEFILE. To change an existing initialization, or create a new one, a new <LISP>USERNAMEFILE must be written. This is accomplished by loading the file <LISP>USERNAMES, editing usernamelst, and then performing makeusernames[], which will create new versions of both USERNAMEFILE and USERNAMES. (Note that the person performing this operation must therefore either be connected to the LISP directory, or have write access to it.)

usernamelst is a list of elements of the form (username firstname T . forms), e.g., (TEITELMAN WARREN T (CHANGESLICE 100) (SETQ DWIMWAIT 5)). cadr of the list is used in the greeting message. cdddr is a list of forms that are evaluated.

usernamelst can be edited just like any other list, e.g., with editv. The file USERNAMEFILE, created by makeusernames, contains usernamelst along with an index block which contains for each user on usernamelst the address in the file

---

80 This message can be suppressed by setting the variable sysoutgag to NIL (initially T). Alternatively, sysoutgag can be a list, in which case it is evaluated (if the user names are different), and thus the user can print his own message.

(i.e., byte position) of the start of his entry. greet then simply does an sfptr and a read.

If usernamelst contains an element for which the username is NIL, i.e., an element of the form (NIL . forms), this is interpreted to mean that forms are evaluated *regardless* of user name. This feature can be used to "patch" an INTERLISP system when a bug is found, or to change some default for INTERLISP at a particular site, e.g., turn off DWIM, perform lowercase[T], etc. Individual user initialization will still be performed following this system initialization.

Index for Section 22

Page Numbers

Page Numbers

SECTION 23[1]

CLISP - CONVERSATIONAL LISP

23.1 Introduction

The syntax of LISP is very simple, in the sense that it can be described concisely, but not in the sense that LISP programs are easy to read or write! This simplicity of syntax is achieved by, and at the expense of, extensive use of explicit structuring, namely grouping through parenthesesization. Unlike many languages, there are no reserved words in LISP such as IF, THEN, AND, OR, FOR, DO, BEGIN, END, etc., nor reserved characters like +, -, *, /, =, ←, etc.[2] This eliminates entirely the need for parsers and precedence rules in the LISP interpreter and compiler, and thereby makes program manipulation of LISP programs straightforward. In other words, a program that "looks at" other LISP programs does not need to incorporate a lot of syntactic information. For example, a LISP interpreter can be written in one or two pages of LISP code ([McC1], pp. 70-71). It is for this reason that LISP is by far the most suitable, and frequently used, programming language for writing programs that deal with other programs as data, e.g., programs that analyze, modify, or construct other programs.

---

1 CLISP was designed and implemented by W. Teitelman. It is discussed in [Tei5].

2 except for parentheses (and period), which are used for indicating structure, and space and end-of-line, which are used for delimiting identifiers.

However, it is precisely this same simplicity of syntax that makes LISP programs difficult to read and write (especially for beginners). 'Pushing down' is something programs do very well, and people do poorly. As an example, consider the following two 'equivalent' sentences:

> "The rat that the cat that the dog that I owned chased caught ate the cheese."
>
> versus
>
> "I own the dog that chased the cat that caught the rat that ate the cheese."

Natural language contains many linguistic devices such as that illustrated in the second sentence above for minimizing embedding, because embedded sentences are more difficult to grasp and understand than equivalent non-embedded ones (even if the latter sentences are somewhat longer). Similarly, most high level programming languages offer syntactic devices for reducing apparent depth and complexity of a program: the reserved words and infix operators used in ALGOL-like languages simultaneously delimit operands and operations, and also convey meaning to the programmer. They are far more intuitive than parentheses. In fact, since LISP uses parentheses (i.e. lists) for almost all syntactic forms, there is very little information contained in the parentheses for the person reading a LISP program, and so the parentheses tend mostly to be ignored: the meaning of a particular LISP expression for people is found almost entirely in the *words*, not in the structure. For example, the following expression

```
(COND (EQ N 0) 1) (T TIMES N FACTORIAL ((SUB1 N)))
```

is recognizable as FACTORIAL even though there are five misplaced or missing parentheses. Grouping words together in parentheses is done more for LISP's benefit, than for the programmer's.

CLISP is designed to make INTERLISP programs easier to read and write by

permitting the user to employ various infix operators, IF-THEN-ELSE statements, FOR-DO-WHILE-UNLESS-FROM-TO-etc. expressions, which are automatically converted to equivalent INTERLISP expressions when they are first interpreted. For example, FACTORIAL could be written in CLISP:

```
(IF N=0 THEN 1 ELSE N*(FACTORIAL N-1))
```

Note that this expression would become an equivalent COND after it had been interpreted once, so that programs that might have to analyze or otherwise process this expression could take advantage of the simple syntax.

There have been similar efforts in other LISP systems, most notably the MLISP language at Stanford [Smi1]. CLISP differs from these in that it does not attempt to *replace* the LISP syntax so much as to *augment* it. In fact, one of the principal criteria in the design of CLISP was that users be able to freely intermix LISP and CLISP without having to identify which is which. Users can write programs, or type in expressions for evaluation, in LISP, CLISP, or a mixture of both. In this way, users do not have to learn a whole new language and syntax in order to be able to use selected facilities of CLISP when and where they find them useful.

CLISP is implemented via the error correction machinery in INTERLISP (see Section 17). Thus, any expression that is well-formed from INTERLISP's standpoint will never be seen by CLISP (i.e., if the user defined a function IF, he would effectively turn off that part of CLISP). This means that interpreted programs that do not use CLISP constructs do not pay for its availability by slower execution time. In fact, the INTERLISP interpreter does not 'know' about CLISP at all. It operates as before, and when an erroneous form is encountered, the interpreter calls an error routine which in turn invokes the Do-What-I-Mean (DWIM) analyzer which contains CLISP. If the expression in question turns out to be a CLISP construct, the equivalent

INTERLISP form is returned to the interpreter. In addition, the original CLISP expression, is modified so that it *becomes* the correctly translated INTERLISP form. In this way, the analysis and translation are done only once.

Integrating CLISP into the INTERLISP system (instead of, for example, implementing it as a separate preprocessor) makes possible Do-What-I-Mean features for CLISP constructs as well as for pure LISP expressions. For example, if the user has defined a function named GET-PARENT, CLISP would know not to attempt to interpret the form (GET-PARENT) as an arithmetic infix operation. (Actually, CLISP would never get to see this form, since it does not contain any errors.) If the user mistakenly writes (GET-PRAENT), CLISP would know he meant (GET-PARENT), and not (DIFFERENCE GET PRAENT), by using the information that PRAENT is not the name of a variable, and that GET-PARENT is the name of a user function whose spelling is "very close" to that of GET-PRAENT. Similarly, by using information about the program's environment not readily available to a preprocessor, CLISP can successfully resolve the following sorts of ambiguities:

1) (LIST X*FACT N), where FACT is the name of a variable, means (LIST (X*FACT) N).

2) (LIST X*FACT N), where FACT is *not* the name of a variable but instead is the name of a function, means (LIST X*(FACT N)), i.e., N is FACT's argument.

3) (LIST X*FACT(N)), FACT the name of a function (and not the name of a variable), means (LIST X*(FACT N)).

4) cases (1),(2) and (3) with FACT misspelled!

The first expression is correct both from the standpoint of CLISP syntax and

semantics and the change would be made without the user being notified. In the other cases, the user would be informed or consulted about what was taking place. For example, to take an extreme case, suppose the expression (LIST X*FCCT N) were encountered, where there was both a function named FACT and a variable named FCT. The user would first be asked if FCCT were a misspelling of FCT. If he said YES, the expression would be interpreted as (LIST (X*FCT) N).[3] If he said NO, the user would be asked if FCCT were a misspelling of FACT, i.e., if he intended X*FCCT N to mean X*(FACT N). If he said YES to this question, the indicated transformation would be performed. If he said NO, the system would then ask if X*FCCT should be treated as CLISP, since FCCT is not the name of a (bound) variable.[4] If he said YES, the expression would be transformed, if NO, it would be left alone, i.e., as (LIST X*FCCT N). Note that we have not even considered the case where X*FCCT is itself a misspelling of a variable name, e.g., a variable named XFCT (as with GET-PRAENT). This sort of transformation would be considered after the user said NO to X*FCCT N -> X*(FACT N). The graph of the possible interpretations for (LIST X*FCCT N) where FCT and XFCT are the names of variables, and FACT is the name of a function, is shown in Figure 23-1 below.

---

[3] Through this discussion, we speak of CLISP or DWIM asking the user. Actually, if the expression in question was typed in by the user for immediate execution, the user is simply informed of the transformation, on the grounds that the user would prefer an occasional misinterpretation rather than being continuously bothered, especially since he can always retype what he intended if a mistake occurs, and ask the programmer's assistant to UNDO the effects of the mistaken operations if necessary. For transformations on expressions in user programs, the user can inform CLISP whether he wishes to operate in CAUTIOUS or TRUSTING mode. In the former case (most typical) the user will be asked to approve transformations, in the latter, CLISP will operate as it does on type-in, i.e., perform the transformation after informing the user.

[4] This question is important because many INTERLISP users already have programs that employ identifiers containing CLISP operators. Thus, if CLISP encounters the expression A/B in a context where either A or B are not the names of variables, it will ask the user if A/B is intended to be CLISP, in case the user really does have a free variable named A/B.

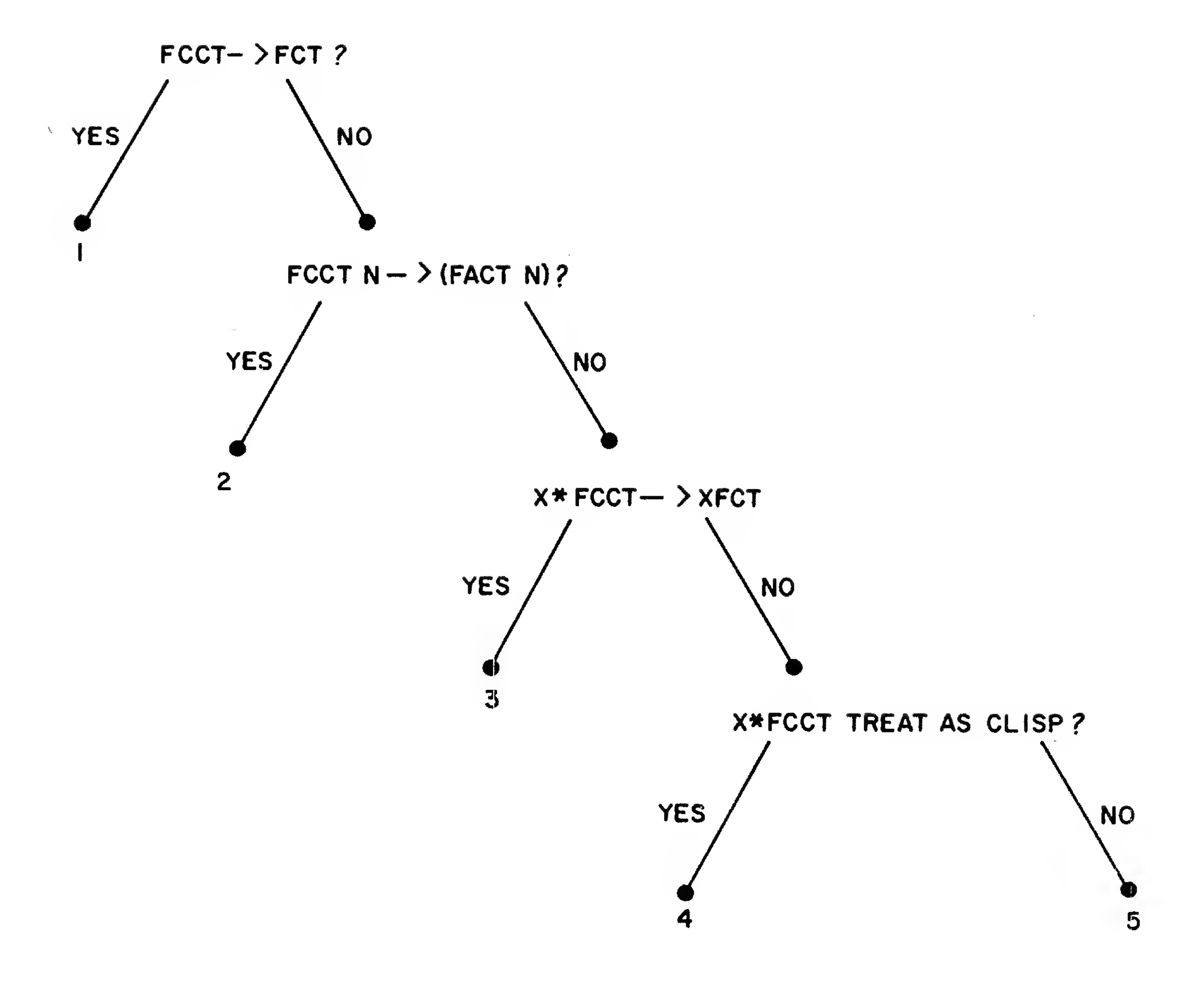

FCCT- >FCT ?
YES
NO
1
FCCT N - > (FACT N)?
YES
NO
2
X* FCCT— > XFCT
YES
NO
3
X*FCCT TREAT AS CLISP ?
YES
NO
4
5

FIGURE 23-1

The final states for the various terminal nodes shown in the graph are:

1: (LIST (TIMES X FCT) N)

2: (LIST (TIMES X (FACT N)))

3: (LIST XFCT N)

4: (LIST (TIMES X FCCT) N)

5: (LIST X*FCCT N)

CLISP can also handle parentheses errors caused by typing 8 or 9 for '(' or ')'. (On most terminals, 8 and 9 are the lower case characters for '(' and ')', i.e., '(' and '8' appear on the same key, as do ')' and '9'.) For example, if the user writes N*8FACTORIAL N-1, the parentheses error can be detected and fixed before the infix operator * is converted to the INTERLISP function TIMES. CLISP is able to distinguish this situation from cases like N*8*X meaning (TIMES N 8 X), or N*8X, where 8X is the name of a variable, again by using information about the programming environment. In fact, by integrating CLISP with DWIM, CLISP has been made sufficiently tolerant of errors that almost everything can be misspelled! For example, CLISP can successfully translate the definition of FACTORIAL:

```
(IFF N=0 THENN1 ESLE N*8FACTTORIALNN-1)
```

to the corresponding COND, while making 5 spelling corrections and fixing the parenthesis error.[5]

This sort of robustness prevails throughout CLISP. For example, the iterative

---

[5] CLISP also contains a facility for converting from INTERLISP back to CLISP, so that after running the above incorrect definition of FACTORIAL, the user could 'CLISPIFY' the now correct LISP version to obtain (IF N=0 THEN 1 ELSE N*(FACTORIAL N-1)).

statement permits the user to say things like:[6]

```
FOR OLD X FROM M TO N DO (PRINT X) WHILE (PRIMEP X)
```

However, the user can also write OLD (X←M), (OLD X←M), (OLD (X←M)), permute the order of the operators, e.g., DO PRINT X TO N FOR OLD X←M WHILE PRIMEP X, omit either or both sets of parentheses, misspell any or all of the operators FOR, OLD, FROM, TO, DO, or WHILE, or leave out the word DO entirely! And, of course, he can also misspell PRINT, PRIMEP, M or N![7]

CLISP is well integrated into the INTERLISP system. For example, the above iterative statement translates into an equivalent INTERLISP form using PROG, COND, GO, etc.[8] When the interpreter subsequently encounters this CLISP expression, it automatically obtains and evaluates the translation.[9] Similarly, the compiler "knows" to compile the translated form. However, if the user PRETTYPRINTs his program, at the corresponding point in his function, PRETTYPRINT "knows" to print the original CLISP. Similarly, when the user edits his program, the editor keeps the translation invisible to the user. If

---

6 This expression should be self explanatory, except possibly for the operator OLD, which says X is to be the variable of iteration, i.e., the one to be stepped from N to M, but X is not to be rebound. Thus when this loop finishes execution, X will be equal to N+1.

7 In this example, the only thing the user could not misspell is the first X, since it specifies the *name* of the variable of iteration. The other two instances of X could be misspelled.

8
```
(PROG NIL
      (SETQ X M)
  $$LP(COND
        ((OR (IGREATERP X N)
             (NOT (PRIMEP X)))
           (RETURN)))
      (PRINT X)
      (SETQ X (ADD1 X))
      (GO $$LP))
```

9 See page 23.30, for discussion of how translations are stored.

the user modifies the CLISP, the translation is automatically discarded and recomputed the next time the expression is evaluated.

In short, CLISP is not a language at all, but rather a system. It plays a role analagous to that of the programmer's assistant (Section 22). Whereas the programmer's assistant is an invisible intermediary agent between the user's console requests and the INTERLISP executive, CLISP sits between the user's programs and the INTERLISP interpreter.

Only a small effort has been devoted to defining the core syntax of CLISP. Instead, most of the effort has been concentrated on providing a facility which 'makes sense' out of the input expressions using context information as well as built-in and acquired information about user and system programs. It has been said that communication is based on the intention of the speaker to produce an effect in the recipient. CLISP operates under the assumption that what the user said was *intended* to represent a meaningful operation, and therefore tries very hard to make sense out of it. The motivation behind CLISP is not to provide the user with many different ways of saying the same thing, but to enable him to worry less about the *syntactic* aspects of his communication with the system. In other words, it gives the user a new degree of freedom by permitting him to concentrate more on the problem at hand, rather than on translation into a formal and unambiguous language.

23.2 CLISP Syntax

Throughout CLISP, a non-atomic form, i.e., a list, can always be substituted for a variable, and vice versa, without changing the interpretation. For example, if the value of (FOO X) is A, and the value of (FIE Y) is B, then (LIST (FOO X)+(FIE Y)) has the same value as (LIST A+B). Note that the first expression consists of a list of *four* elements: the atom 'LIST', the list

'(FOO X)', the atom '+', and the list '(FIE X)', whereas the second expression, (LIST A+B), consists of a list of only *two* elements: the atom 'LIST' and the atom 'A+B'. Since (LIST (FOO X)+(FIE Y)) is indistinguishable from (LIST (FOO X)_+_(FIE Y)) because spaces before or after parentheses have no effect on the INTERLISP READ program,[10] to be consistent, extra spaces have no effect on atomic operands either. In other words, CLISP will treat (LIST A+_B), (LIST A_+B), and (LIST A_+_B) the same as (LIST A+B).

23.3 Infix Operators

CLISP recognizes the arithmetic infix operators +, -, *, /, and ↑. These are converted to IPLUS, IDIFFERENCE (or in the case of unary minus, IMINUS), ITIMES, IQUOTIENT, and EXPT.[11] The usual precedence rules apply (although these can be easily changed by the user),[12] i.e., * has higher precedence than + so that A+B*C is the same as A+(B*C), and both * and / are lower than ↑ so that 2*X↑2 is the same as 2*(X↑2). Operators of the same precedence group from left to right, e.g., A/B/C is equivalent to (A/B)/C. Minus is binary whenever possible, i.e., except when it is the first operator in a list, as in (-A) or (-A), or when it immediately follows another operator, as in A*-B.[13] [14]

---

[10] CLISP does not use its own special READ program because this would require the user to explicitly identify CLISP expressions, instead of being able to intermix INTERLISP and CLISP.

[11] The I in IPLUS denotes integer arithmetic, i.e., IPLUS converts its arguments to integers, and returns an integer value. INTERLISP also contains floating point arithmetic functions as well as mixed arithmetic functions (see Section 13). CLISP contains a facility for declaring which type of arithmetic is to be used, either by making a global declaration, or by separate declarations about individual functions or variables. See section on declarations, page 23.33.

[12] The complete order of precedence for CLISP operators is given in Figure 23-2, page 23.15.

[13] There are some do-what-I-mean features associated with Unary minus, as in (LIST -X Y). See section on operation, page 23.64.

Note that grouping with parentheses can always be used to override the normal precedence grouping, or when the user is not sure how a particular expression will parse.

CLISP also recognizes as infix operators =, GT, LT, GE, and LE,[15] as well as various predicates, e.g., MEMBER, AND, OR, EQUAL, etc.[16] AND is higher than OR, e.g., (X OR Y AND Z) is the same as (X OR (Y AND Z)), and both AND and OR are lower than the other infix operators, e.g., (X AND Y EQUAL Z) is the same as (X AND (Y EQUAL Z)). All of the infix predicates have lower precedence than INTERLISP forms, i.e., (FOO X GT FIE Y) is the same as ((FOO X) GT (FIE Y)), since it is far more common to apply a predicate to two forms, than to use a Boolean as an argument to a function, e.g. (FOO (X GT (FIE Y))). However, again, the user can easily change this.

Note that only single character operators, e.g. +, ←, =, etc., can appear in the interior of an atom. All other operators must be set off from identifiers with spaces. For example, XLTY *will not be recognized as CLISP.*[17]

---

[14] Note that + in front of a number will disappear when the number is read, e.g., (FOO X +2) is indistinguishable from (FOO X 2). This means that (FOO X +2) will not be interpreted as CLISP, or be converted to (FOO (IPLUS X 2)). Similarly, (FOO X -2) will not be interpreted the same as (FOO X-2). To circumvent this, always type a space between the + or - and a number if an infix operator is intended, e.g., write (FOO X + 2).

[15] Greater Than, Less Than, Greater than or Equal to, and Less than or Equal to, respectively. GT, LT, GE, and LE are all affected by the same declarations as + and *, with the initial default to use IGREATERP and ILESSP.

[16] Currently the complete list is MEMBER, MEMB, FMEMB, ILESSP, IGREATERP, LESSP, GREATERP, FGTP, EQ, NEQ, EQP, EQUAL, OR, and AND. New infix operators can be easily added, as described in the section on CLISP internal conventions, page 23.68.

[17] In some cases, DWIM will be able to diagnose this situation as a run-on spelling error, in which case after the atom is split apart, CLISP will be able to perform the indicated transformation.

* * *

: is an infix operator used in CLISP for extracting substructures from lists,[18] e.g., X:3 specifies the 3rd element of X, (FOO Y)::2 specifies the second tail of (FOO Y), i.e., (CDDR (FOO Y)), and Z:1:2 is the second element of the first element of Z, or (CADAR Z). Negative numbers may be used to indicate position counting from the end of a list, e.g., X:-1 is the last element of X, or (CAR (LAST X)), X::-1 is the last *tail*, i.e., (LAST X).[19]

* * *

← is used to indicate assignment, e.g., X←Y translates to (SETQ X Y).[20] [21] In conjunction with : and ::, ← can also be used to perform a more general type of assignment, namely one involving structure modification. For example, X:2←Y means make the second element of X *be* Y, in INTERLISP terms

---

[18] The record facility, page 23.48, provides another way of extracting substructures by allowing the user to assign names to the various parts of the structure and then retrieve from or store into the corresponding structure by name. The pattern match facility, page 23.36, also can be used to extract substructure. : is also used to indicate both record and pattern match operations.

[19] The interpretation of negative numbers can be explained neatly in terms of edit commands: :-n returns what would be the current expression after executing the command -n, and ::-n returns what would be the current expression after executing -n followed by UP.

[20] If x does not have a value, and is not the name of one of the bound variables of the function in which it appears, spelling correction is attempted. However, since this may simply be a case of assigning an initial value to a new free variable, DWIM will always ask for approval before making the correction.

[21] Note that an atom of the form X←Y, appearing at the top level of a PROG, will *not* be recognized as an assignment statement because it will be interpreted as a PROG label by the INTERLISP interpreter, and therefore will not cause an error, so DWIM and CLISP will never get to see it. Instead, one must write (X←Y).

(RPLACA (CDR X) Y).[22] [23] Negative numbers can also be used, e.g., X:-2←Y.[24] ← is also used to indicate assignment in record operations, page 23.48, and pattern match operations, page 23.36.

← has different precedence on the left from on the right. On the left, ← is a "tight" operator, i.e., high precedence, so that A+B←C is the same as A+(B←C). On the right, ← has broader scope so that A←B+C is the same as A←(B+C).

On typein, $←form (alt-mode←form) is equivalent to set the "last thing mentioned".[25] For example, immediately after examining the value of LONGVARIABLENAME, the user could set it by typing $← followed by a form.

23.4 Prefix Operators

CLISP recognizes ' and ~ as prefix operators. ' means QUOTE when it is the first character in an identifier, and is ignored when it is used in the interior of an identifier. Thus, X='Y means (EQ X (QUOTE Y)), but X=CAN'T means (EQ X CAN'T), *not* (EQ X CAN) followed by (QUOTE T). This enables users to have variable and function names with ' in them (so long as the ' is not the first character).

---

[22] Note that the *value* of this operation is the value of rplaca, which is the corresponding *node*.

[23] The user can indicate he wants /rplaca and /rplacd used (undoable version of rplaca and rplacd, see Section 22), or frplaca and frplacd (fast versions of rplaca and rplacd, see Section 5), by means of declarations (page 23.33). The initial default is for rplaca and rplacd.

[24] which translates to (RPLACA (NLEFT X 2) Y).

[25] i.e. is equivalent to (SETQ lastword form). See Section 17.

Following ', all operators are ignored for the rest of the identifier, e.g., '*A means (QUOTE *A), and 'X=Y means (QUOTE X=Y), not (EQ (QUOTE X) Y).[26]

On typein, '$ (i.e. 'alt-mode) is equivalent to (QUOTE value-of-lastword) (see Section 17). For example, after calling prettyprint on LONGFUNCTION, the user could move its definition to FOO by typing (MOVD '$ 'FOO).[27]

~ means NOT. ~ can negate a form, as in ~(ASSOC X Y), or ~X, or negate an infix operator, e.g., (A ~GT B) is the same as (A LEQ B). Note that ~A=B means (EQ (NOT A) B).

---

[26] To write (EQ (QUOTE X) Y), one writes Y='X, or 'X_=Y. This is one place where an extra space does make a difference.

[27] *Not* (MOVD $ 'FOO), which would be equivalent to (MOVD LONGFUNCTION 'FOO), and would (probably) cause a U.B.A. LONGFUNCTION error, nor MOVD($ FOO), which would actually move the definition of $ to FOO, since DWIM and the spelling corrector would never be invoked.

Order of Precedence of CLISP operators

'

:

← (left precedence) [28]

- (unary), ~ [29]

↑

*, /

+, - (binary)

← (right precedence)

=

INTERLISP forms

LT, GT, EQUAL, MEMBER, etc.

AND

OR

IF, THEN, ELSEIF, ELSE

iterative statement operators

Figure 23-2

---

[28] ← has a different left and right precedence, e.g., A+B←C+D is the same as A+(B←(C+D)). In other words, ← has minimal scope on the left and maximal scope on the right.

[29] When ~ negates an operator, e.g., ~=, ~LT, the two operators are treated as a single operator whose precedence is that of the second operator. When ~ negates a function, e.g., (~FOO X Y), it negates the whole form, i.e., (~(FOO X Y)).

23.5 Constructing Lists - the <,> operators[30]

Angle brackets are used in CLISP to indicate list construction. The appearance of a '<' corresponds to a '(' and indicates that a list is to be constructed containing all the elements up to the corresponding '>'. For example, <A B <C>> translates to (LIST A B (LIST C)). ! can be used to indicate that the next expression is to be inserted in the list as a *segment*, e.g., <A B ! C> translates to (CONS A (CONS B C)) and <! A ! B C> to (APPEND A B (LIST C)). !! is used to indicate that the next expression is to be inserted as a segment, and furthermore, all list structure to its right in the angle brackets is to be physically attached to it, e.g., <!! A B> translates to (NCONC1 A B), and <!!A !B !C> to (NCONC A (APPEND B C)).[31] [32] Note that <, !, !!, and > need not be separate atoms, for example, <A B ! C> may be written equally well as < A B !C >. Also, arbitrary INTERLISP or CLISP forms may be used within angle brackets. For example, one can write <FOO←(FIE X) ! Y> which translates to (CONS (SETQ FOO (FIE X)) Y). CLISPIFY converts expressions in cons, list, append, nconc, nconc1, /nconc, and /nconc1 into equivalent CLISP expressions using <, >, !, and !!.

Note: angle brackets differ from other CLISP operators in that they act more like brackets than operators. For example, <A B 'C> translates to (LIST A B (QUOTE C)) even though following ', all *operators* are ignored for the

---

[30] The <,> operator was written by P.C. Jackson.

[31] Not (NCONC (APPEND A B) C), which would have the same value, but would attach C to B, and not attach either to A.

[32] The user can indicate /nconc or /nconc1 be used instead of nconc and nconc1 by declarations.

rest of the identifier.[33] Note however that <A B '_C> D> is equivalent to (LIST A B (QUOTE C>) D).

23.6 IF, THEN, ELSE

CLISP translates expressions employing IF|THEN|ELSEIF|ELSE into equivalent conditional expressions. The segment between IF|ELSEIF and the next THEN corresponds to the predicate of a COND clause, and the segment between THEN and the next ELSE|ELSEIF as the consequent(s). ELSE is the same as ELSEIF T THEN.

IF, THEN, ELSE, and ELSEIF are of lower precedence than all infix and prefix operators, as well as INTERLISP forms, so that parentheses can be omitted between IF|ELSEIF, and THEN. For example, (IF FOO X Y THEN --) is equivalent to (IF (FOO X Y) THEN --).[34] Similarly, CLISP treats (IF X THEN FOO X Y ELSE --) as equivalent to (IF X THEN (FOO X Y) ELSE --) because it does not 'make sense' to evaluate a variable for effect. In other words, even if FOO were also the name of a variable, (COND (X FOO X Y)) doesn't make sense. Essentially, CLISP determines whether the segment between THEN and the next ELSE|ELSEIF corresponds to one form or several and acts accordingly.[35] Thus, (IF -- THEN (FOO X) Y ELSE --) corresponds to a clause with two consequents. Similarly, (IF -- THEN FOO←X Y ELSE --) corresponds to a clause with two

---

[33] Only if a previous unmatched < has been seen, e.g. (PRINT 'A>B) will print the atom A>B.

[34] If FOO is the name of a variable, IF FOO THEN -- is translated as (COND (FOO --)) even if FOO is also the name of a function. If the functional interpretation is intended, FOO should be enclosed in parentheses, e.g., IF (FOO) THEN --. Similary for IF -- THEN FOO ELSEIF --.

[35] occasionally interacting with the user to resolve ambiguous cases.

consequents, and is equivalent to (IF -- THEN (FOO←X) Y ELSE --).[36]

23.7 Iterative Statements

The following is an example of a CLISP iterative statement:

```
(WHILE X←(READ)~='STOP DO (PRINT (EVAL X)))
```

This statement says "READ an expression and set X to it. If X is not equal to the atom STOP, then evaluate X, print the result, and iterate."[37]

The i.s. (iterative statement) in its various forms permits the user to specify complicated iterative statements in a straightforward and visible manner. Rather than the user having to perform the mental transformations to an equivalent INTERLISP form using PROG, MAPC, MAPCAR, etc., the system does it for him. The goal was to provide a robust and tolerant facility which could "make sense" out of a wide class of iterative statements. Accordingly, the user should not feel obliged to read and understand in detail the description of each operator given below in order to use iterative statements.

Currently, the following i.s. operators are implemented: FOR, BIND, OLD, IN, ON, FROM, TO, BY, WHEN, WHILE, UNTIL, UNLESS, COLLECT, JOIN, DO, SUM, COUNT, ALWAYS, NEVER, THEREIS, AS, FIRST, FINALLY, EACHTIME. Their function is

---

[36] To write the equivalent of a singleton cond clause, i.e., a clause with a predicate but no consequent, write either nothing following the THEN, or omit the THEN entirely, e.g., (IF (FOO X) THEN ELSEIF --) or (IF (FOO X) ELSEIF --), meaning if (FOO X) is not NIL, it is the value of the cond.

[37] The statement translates to:
```
(PROG NIL $$LP (COND ((EQ (SETQ X (READ)) (QUOTE STOP)) (RETURN)))
(PRINT (EVAL X)) (GO $$LP))
```

explained below. New operators can be defined as described on page 23.28. Misspellings of operators are recognized and corrected. The order of appearance of operators is not important;[38] CLISP scans the entire statement before it begins to construct the equivalent INTERLISP form.

DO form — specifies what is to be done at each iteration. DO with no other operator specifies an infinite loop. If some explicit or implicit terminating condition is specified, the value of the i.s. is NIL. Translate to MAPC or MAP whenever possible.

COLLECT form — like DO, except specifies that the value of <u>form</u> at each iteration is to be collected in a list, which is returned as the value of the i.s. when it terminates. Translates to MAPCAR or MAPLIST whenever possible.

JOIN form — like DO, except that the values are NCONCed. Translates to MAPCONC or MAPCON whenever possible.[39]

SUM form — like DO, except specifies that the values of <u>form</u> at each iteration be added together and returned as the value of the i.s., e.g. (FOR I FROM 1 TO 5 SUM I↑2) is equal to 1+4+9+16+25.[40]

---

[38] DWIM and CLISP are invoked on iterative statements because <u>car</u> of the i.s. is not the name of a function, and hence generates an error. If the user defines a function by the same name as an i.s. operator, e.g. WHILE, TO, etc., the operator will no longer have the CLISP interpretation when it appears as <u>car</u> of a form, although it will continue to be treated as an i.s. operator if it appears in the interior of an i.s.

[39] /NCONC, /MAPCONC, and /MAPCON are used when the declaration UNDOABLE is in effect.

[40] <u>iplus</u>, <u>fplus</u>, or <u>plus</u> will be used for the translation depending on the declarations in effect.

COUNT pred — like DO, except counts number of times that pred is true, and returns that count as its value.

ALWAYS pred — like DO, except returns T if the value of pred is non-NIL for all iterations (returns NIL as soon as the value of pred is NIL), e.g. (FOR X IN Y ALWAYS (ATOM X)) is the same as (EVERY Y (FUNCTION ATOM)).

NEVER pred — like ALWAYS, except returns T if the value of pred is *never* true, i.e. NEVER pred is the same as ALWAYS ~pred.

THEREIS pred — returns the first value of the i.v. for which pred is non-NIL, e.g. (FOR X IN Y THEREIS NUMBERP) returns the first number in Y, and is equivalent to (CAR (SOME Y (FUNCTION NUMBERP))).[41]

DO, COLLECT, JOIN, SUM, ALWAYS, NEVER, and THEREIS are examples of a certain kind of i.s. operator called an i.s.type. The i.s.type specifies what is to be done at each iteration. Each i.s. must have one and only one i.s.type. The function i.s.type, page 23.28, provides a means of defining additional i.s.types.

FOR var — specifies the variable of iteration, or i.v., which is used in conjunction with IN, ON, FROM, TO, and BY. The variable is rebound for the scope of the i.s., except when modified by OLD as described below.

---

[41] THEREIS returns the i.v. instead of the tail (as does the function some) in order to provide an interpretation consistent with statements such as (FOR I FROM 1 TO 10 THEREIS --), where there is no tail. Note that (SOME Y (FUNCTION NUMBERP)) is equivalent to (FOR X ON Y THEREIS (NUMBERP (CAR X))).

FOR vars — vars a list of variables, e.g., FOR (X Y Z) IN --. The first variable is the i.v., the rest are dummy variables. See BIND below.

OLD var — indicates var is *not* to be rebound, e.g., (FOR OLD X FROM 1 TO N DO -- UNTIL --),

BIND var, vars — used to specify dummy variables, e.g., FOR (X Y Z) IN -- is equivalent to FOR X BIND (Y Z) IN --. BIND can be used without FOR. For example, in the i.s. shown on page 23.18, X could be made local by writing (BIND X WHILE X←(READ)~='STOP...).

Note: FOR, OLD, and BIND variables can be initialized by using ←, e.g., (FOR OLD (X←form) BIND (Y←form)...).

IN form — specifies that the i.s. is to iterate down a list with the i.v. being reset to the corresponding element at each iteration. For example, FOR X IN Y DO -- corresponds to (MAPC Y (FUNCTION (LAMBDA (X) --))). If no i.v. has been specified, a dummy is supplied, e.g., IN Y COLLECT CADR is equivalent to (MAPCAR Y (FUNCTION CADR)).

ON form — same as IN except that the i.v. is reset to the corresponding *tail* at each iteration. Thus IN corresponds to MAPC, MAPCAR, and MAPCONC, while ON corresponds to MAP, MAPLIST, and MAPCON.

IN OLD var — specifies that the i.s. is to iterate down var, with var itself being reset to the corresponding tail at each iteration, e.g., after (FOR X IN OLD L DO -- UNTIL --) finishes, L will be some tail of its original value.

IN OLD (var←form) same as IN OLD var, except var is first set to value of form.

ON OLD var same as IN OLD var except the i.v. is reset to the current value of var at each iteration, instead of to car[var].

ON OLD (var←form) same as ON OLD var, except var is first set to value of form.

WHEN pred provides a way of excepting certain iterations. For example, (FOR X IN Y COLLECT X WHEN NUMBERP X) collects only the elements of Y that are numbers.

UNLESS pred same as WHEN except for the difference in sign, i.e., WHEN Z is the same as UNLESS ~Z.

WHILE pred provides a way of terminating the i.s. WHILE pred evaluates pred *before* each iteration, and if the value is NIL, exits.

UNTIL pred Same as WHILE except for difference in sign, i.e., WHILE PRED is equivalent to UNTIL ~PRED.

UNTIL n n a number, equivalent to UNTIL (i.v. GT n).

FROM form is used to specify an initial value for a numerical i.v. The i.v. is automatically incremented by 1 after each iteration (unless BY is specified). If no i.v. has been specified, a dummy i.v. is supplied and initialized, e.g., (COLLECT SQRT FROM 2 TO 5) returns (1.414 1.732 2.0 2.236).

TO form is used to specify the final value for a numerical i.v. If

FROM is not specified, the i.v. is initialized to 1. If no i.v. has been specified, a dummy i.v. is supplied and initialized. If BY is not specified, the i.v. is automatically incremented by 1 after each iteration.[42] When the i.v. is definitely being *incremented*, i.e. either BY is not specified, or its operand is a positive number, the i.s. terminates when the i.v. exceeds the value of form (which is reevaluated each iteration) e.g., (FOR X FROM 1 TO 10 --), is equivalent to (FOR X FROM 1 UNTIL (X GT 10) --).

Similarly, when the i.v. is definitely being decremented the i.s. terminates when the i.v. becomes *less* than the value of form (see description of BY).

BY x (with IN/ON) If IN or ON have been specified, the value of x determines the *tail* for the next iteration, which in turn determines the value for the i.v. as described earlier, i.e. the new i.v. is car of the tail for IN, the tail itself for ON. In conjunction with IN, the user can refer to the current tail within x by using the i.v., e.g. (FOR Z IN L BY (CDDR Z) ...). At translation time, the name of the internal variable which holds the value of the current tail is substituted for the i.v. throughout x. For example, (FOR X IN Y BY (CDR (MEMB 'FOO (CDR X))) COLLECT X) specifies that after each iteration, cdr of the current tail is to be searched for the atom FOO, and (cdr of) this latter tail to be used for the next iteration.

---

[42] except when both the operands to TO and FROM are numbers, and TO's operand is less than FROM's operand, e.g. FROM 10 TO 1, in which case the i.v. is decremented by 1 after each iteration. In this case, the i.s. terminates when the i.v. becomes *less* than the value of form.

BY x (without IN/ON) If IN or ON have not been used, BY specifies how the i.v. itself is reset at each iteration. If FROM or TO have been specified, the i.v. is known to be numerical, so the new i.v. is computed by adding the value of x (which is reevaluated each iteration) to the current value of the i.v., e.g., (FOR N FROM 1 TO 10 BY 2 COLLECT N) makes a list of the first five odd numbers.

If x is a positive number,[43] the i.s. terminates when the value of the i.v. *exceeds* the value of TO's operand. If x is a negative number, the i.s. terminates when the value of the i.v. becomes *less* than TO's operand, e.g. (FOR I FROM N TO M BY -2 UNTIL (I LT M) ...). Otherwise, the terminating condition for each iteration depends on the value of x for that iteration: if x < 0, the test is whether the i.v. is less than TO's operand, if x > 0 the test is whether the i.v. exceeds TO's operand, otherwise if x=0, the i.s. terminates unconditionally.[44]

If FROM or TO have not been specified, the i.v. is simply reset to the value of x after each iteration, e.g. (FOR I FROM N BY 2 ...) is equivalent to (FOR I←N BY (IPLUS I 2) ...).

---

[43] x itself, not its value, which in general CLISP would have no way of knowing in advance.

[44] A temporary variable is used so that x is only evaluated once. However, code for TO's operand appears twice in the translation, even though it is evaluated only once.

FIRST form — form is evaluated once before the first iteration, e.g. (FOR X Y Z IN L -- FIRST (FOO Y Z)), and FOO could be used to initialize Y and Z.

FINALLY form — form is evaluated after the i.s. terminates.[45] For example, (FOR X IN L BIND Y←0 DO (IF ATOM X THEN Y←Y+1) FINALLY (RETURN Y)) will return the number of atoms in L.

EACHTIME form — form is evaluated at the beginning of each iteration before, and regardless of, any testing. For example, consider (FOR I FROM 1 TO N DO (... (FOO I) ...) UNLESS (... (FOO I) ...) UNTIL (... (FOO I) ...)). The user might want to set a temporary variable to the value of (FOO I) in order to avoid computing it three times each iteration. However, without knowing the translation, he would not know whether to put the assignment in the operand to DO, UNLESS, or UNTIL, i.e. which one would be executed first. He can avoid this problem by simply writing EACHTIME J←(FOO I).

AS var — is used to specify an iterative statement involving more than one iterative variable, e.g. (FOR X IN Y AS U IN V DO --) corresponds to map2c. The i.s. terminates when any of the terminating conditions are met, e.g. (FOR X IN Y AS I FROM I TO 10 COLLECT X) makes a list of the first ten elements of Y, or however many elements there are on Y if less than 10.

---

[45] Except in the case of termination due to the appearance of a RETURN in some operand. See page 23.26. Thus in (FOR X IN Y THEREIS NUMBERP FINALLY --) the FINALLY operand would be evaluated if Y were exhausted, but not if a number was found.

The operand to AS, <u>var</u>, specifies the new i.v. For the remainder of the i.s., or until another AS is encountered, all operators refer to the new i.v. For example, (FOR I FROM I TO N1 AS J FROM 1 TO N2 BY 2
AS K FROM N3 TO 1 BY -1 --) terminates when I exceeds N1, or J exceeds N2, or K becomes less than 1. After each iteration, I is incremented by 1, J by 2, and K by -1.

<u>Miscellaneous</u>

1. Lowercase versions of all i.s. operators are equivalent to the uppercase, e.g., (for X in Y ...).
2. Each i.s. operator is of lower precedence than all INTERLISP forms, so parentheses around the operands can be omitted, and will be supplied where necessary, e.g., BIND (X Y Z) can be written BIND X Y Z, OLD (X←form) as OLD X←form, WHEN (NUMBERP X) as WHEN NUMBERP X, etc.
3. RETURN or GO may be used in any operand. RETURN means return from the i.s. (with the indicated value), *not* from the function in which the i.s appears. GO refers to a label elsewhere in the function in which the i.s. appears, except for (GO ITERATE), which means transfer control to the iterate portion of the loop, i.e. that part that resets the tail, increments the counter, or whatever, in preparation for the next iteration. The appropriate label will be substituted for ITERATE.
4. In the case of FIRST, FINALLY, EACHTIME, or one of the i.s.types, e.g. DO, COLLECT, SUM, etc., the operand can consist of more than one form, e.g., COLLECT (PRINT X:1) X:2, in which case a PROGN is supplied.
5. Each operand can be the name of a function, in which case it is applied to

the (last) i.v.,[46 47 48] e.g., FOR X IN Y DO PRINT WHEN NUMBERP, is the same as FOR X IN Y DO (PRINT X) WHEN (NUMBERP X). Note that the i.v. need not be explicitly specified, e.g., IN Y DO PRINT WHEN NUMBERP will work.

Errors in Iterative Statements

An error will be generated and an appropriate diagnostic printed if any of the following conditions hold:

1. Operator with null operand, i.e. two adjacent operators, as in FOR X IN Y UNTIL DO --
2. Operand consisting of more than one form (except as operand to FIRST, FINALLY, or one of the i.s.types), e.g., FOR X IN Y (PRINT X) COLLECT --.
3. Same operator appears twice.
4. Both IN and ON used on same i.v.
5. FROM or TO used with IN or ON on same i.v.
6. More than one i.s.type, e.g. a DO and a SUM.

In 3, 4, or 5, an error is not generated if an intervening AS occurs.

If an error occurs, the i.s. is left unchanged.

---

[46] For i.s.types, e.g. DO, COLLECT, JOIN, the function is always applied to the first i.v. in the i.s., whether explicity named or not. For example, (IN Y AS I FROM 1 TO 10 DO PRINT) prints elements on Y, not integers between 1 and 10.

[47] Note that this feature does not make much sense for FOR, OLD, BIND, IN, or ON, since they "operate" before the loop starts, when the i.v. may not even be bound.

[48] In the case of BY in conjunction with IN, the function is applied to the current *tail* e.g., FOR X IN Y BY CDDR ..., is the same as FOR X IN Y BY (CDDR X)... See page 23.23.

If no DO, COLLECT, JOIN or any of the other i.s.types are specified, CLISP will first attempt to find an operand consisting of more than one form, e.g., FOR X IN Y _(PRINT X)_ WHEN ATOM X, and in this case will insert a DO after the first form. (In this case, condition 2 is not considered to be met, and an error is not generated.) If CLISP cannot find such an operand, a warning message is printed: NO DO, COLLECT, OR JOIN: followed by the i.s. However, the i.s. is still translated, e.g. (WHILE form) can be used to execute _form_ repeatedly until its value is NIL.

Similarly, if no terminating condition is detected, i.e. no IN, ON, WHILE, UNTIL, TO, or a RETURN or GO, a warning message is printed: POSSIBLE NON-TERMINATING ITERATIVE STATEMENT: followed by the i.s. However, since the user may be planning to terminate the i.s. via an error, control-E, or a _retfrom_ from a lower function, the i.s. is still translated.

Defining New Iterative Statement Operators

The i.s.type specifies what is to be done at each iteration, e.g. collecting values on a list, adding numbers, searching for a particular condition, etc. Each i.s. can have one and only one i.s.type. The function _i.s.type_ provides a means of defining new i.s.types.

i.s.type[name;form;init;val] _name_ is the name of the i.s.type. _form_ is the form to be evaluated at each iteration. In _form_ $$VAL can be used to reference the value being assembled, I.V. to reference the current value of the i.v., and BODY to reference the body of the statement, i.e. _name_'s operand.

For example, for COLLECT, _form_ would be (SETQ $$VAL (NCONC1 $$VAL BODY)), for

SUM: ($$VAL←$$VAL+BODY),[49] for NEVER: (AND BODY (RETURN NIL)), and for THEREIS: (AND BODY (RETURN I.V.)).

init specifies the initial value for $$VAL, e.g. for SUM, init is 0. val specifies the value to be returned when the i.s. terminates.

i.s.type is undoable.

Examples:

1) To define RCOLLECT, a version of COLLECT which uses cons instead of nconc1 and then reverses the list of values:

```
i.s.type[RCOLLECT;(SETQ $$VAL (CONS BODY $$VAL));NIL;(DREVERSE $$VAL)]
```

2) To define TCOLLECT, a version of COLLECT which uses tconc:

```
i.s.type[TCOLLECT;(TCONC $$VAL BODY); (CONS); (CAR $$VAL)]
```

3) To define PRODUCT:

```
i.s.type[PRODUCT;($$VAL←$$VAL*BODY);1;$$VAL]
```

i.s.type performs the appropriate modifications to the property list for name, as well as for the lower case version of name, and also updates the appropriate spelling lists.

i.s.type can also be used to define synonyms for all i.s. operators, (not just those that are i.s.types), by calling i.s.type with form an atom, e.g. i.s.type[WHERE;WHEN] makes WHERE be the same as WHEN. Similarly, following

---

[49] $$VAL+BODY is used instead of (IPLUS $$VAL BODY), so that the choice of function used in the translation, i.e. iplus, fplus, or plus, will be determined by the declarations then in effect.

i.s.type[ISTHERE;THEREIS] one can write (ISTHERE ATOM IN Y), and following i.s.type[FIND;FOR] and i.s.type[SUCHTHAT;THEREIS], one can write (FIND X IN Y SUCHTHAT X MEMBER Z).[50]

This completes the description of iterative statements.

23.8 CLISP Translations

The translation of infix operators and IF|THEN|ELSE statements are handled in CLISP by *replacing* the CLISP expression with the corresponding INTERLISP expression, and discarding the original CLISP, because (1) the CLISP expression is easily recomputable (by clispify), [51] and (2) the INTERLISP expressions are simple and straightforward. In addition to saving the space required to retain both the CLISP and the INTERLISP, another reason for discarding the original CLISP is that it may contain errors that were corrected in the course of translation, e.g. the user writes FOO←FOOO:1, N*8FOO X), etc. If the original CLISP were retained, either the user would have to go back and fix these errors by hand, thereby negating the advantage of having DWIM perform these corrections, or else DWIM would have to keep correcting these errors over and over.

Where (1) or (2) are not the case, e.g. with iterative statements, pattern

---

[50] In the current system, WHERE is synonymous with WHEN, SUCHTHAT and ISTHERE with THEREIS, and FIND with FOR.

[51] Note that clispify is sufficiently fast that it is practical for the user to configure his INTERLISP system so that all expressions are automatically clispifyed immediately before they are presented to him. For example, he can define an edit macro to use in place of P which calls clispify on the current expression before printing it. Similarly, he can inform prettyprint to call clispify on each expression before printing it, etc.

matches, record expressions, etc.[52] [53] the original CLISP *is* retained (or a slightly modified version thereof), and the translation is stored elsewhere, usually in clisparray, a hash array. The interpreter automatically checks this array using gethash when given a form car of which is not a function. Similarly, the compiler performs a gethash when given a form it does not recognize to see if it has a translation, which is then compiled instead of the form. Whenever the user *changes* a CLISP expresson by editing it, the editor automatically deletes its translation (if one exists), so that the next time it is evaluated or dwimified, the expression will be retranslated. The function ppt and the edit commands PPT and CLISP: are available for examining translations, see page 23.75. Similarly, if prettytranflg is T, prettyprint will print the translations instead of the corresponding CLISP expression.[54]

If clisparray is NIL,[55] translations are implemented instead by replacing the CLISP expression by an expression of the form (CLISP%_ translation . CLISP-expression),[56] e.g. (FOR X IN Y COLLECT (CAR X))

---

[52] For example, the translation of X:5:3 is (CADDR (CAR (CDDDDR X))), which is difficult to read. Therefore, such expressions are handled by retaining the CLISP and storing the translation elsewhere, as described below.

[53] The handling of translations for IF|THEN|ELSE statements is determined by the value of clispiftranflg. If T, the translations are stored elsewhere, and the (modified) CLISP retained as described below. If NIL, the corresponding COND replaces the IF|THEN|ELSE expression. The initial value of clispiftranflg is NIL.

[54] Note that the user can always examine the translation himself by performing (GETHASH expression CLISPARRAY).

[55] clisparray is initially NIL, and #clisparray is its size. The first time a translation is performed, a hash array of this size is created. Therefore to disable clisparray, both it and #clisparray should be set to NIL.

[56] CLISP%_ is an atom consisting of the six characters C, L, I, S, P, and *space*, which must be preceded by the escape character % in order for it to be included as a part of an identifier. The intent was to deliberately make this atom hard to type so as to make it unlikely that it would otherwise appear in a user's program or data, since the editor and prettyprint treat it very specially, as described above.

would be replaced by

(CLISP%_ (MAPCAR Y (FUNCTION CAR)) FOR X IN Y COLLECT (CAR X)). Both the editor and prettyprint know about CLISP%_ expressions and treat them specially by suppressing the translations: *Prettyprint* prints just the CLISP (unless *prettytranflg*=T, as described below), while the editor makes the translation completely invisible, e.g. if the current expression were the above CLISP%_ expression, F MAPCAR would fail to find the MAPCAR, and (3 ON) would replace IN with ON, i.e. the editor operates as though both the CLISP%_ and the MAPCAR were not there. As with translations implemented via *clisparray*, if the CLISP expression is changed by editing it, the translation is automatically deleted.

CLISP%_ expressions will interpret and compile correctly: CLISP%_ is defined as an nlambda nospread function with an appropriate compiler macro. Note that if the user sets *clisparray* to NIL, he can then break, trace, or advise CLISP%_ to monitor the evaluation of iterative statements, pattern matches, and record operations. This technique will work even if *clisparray* was not NIL at the time the expressions were originally translated, since setting *clisparray* to NIL will effectively delete the translations, thereby causing the CLISP expressions to be retranslated when they are first encountered. Note that if the user only wishes to monitor the CLISP in a certain function, he can accomplish this by embedding its definition in (RESETVAR CLISPARRAY NIL *).

If a CLISP%_ expression is encountered and *clisparray* is *not* NIL, the translation is transferred to the hash array, and the CLISP%_ expression replaced by just the CLISP. Setting *prettytranflg* to CLISP%_ causes *prettyprint* to print CLISP expressions that have been translated in the form of (CLISP%_ translation . CLISP-expression), even if the translation is currently stored in *clisparray*. These two features together provide the user with a way of dumping CLISP expressions together with their translations so that when reloaded (and run or dwimified), the translations will automatically be transferred to *clisparray*.

In summary, if prettytranflg=NIL, only the CLISP is printed (used for producing listings). If prettytranflg=T, only the translation is printed (used for exporting programs to systems that do not provide CLISP, and to examine translations for debugging purposes).[57] If prettytranflg=CLISP%_, an expression of the form (CLISP%_ translation . CLISP) is printed, (used for dumping both CLISP and translations). The preferred method of storing translations is in clisparray, so that if any CLISP%_ expressions are converted while clisparray is not NIL, they will automatically be converted so as to use clisparray. If clisparray=NIL, they will be left alone, and furthermore, new translations will be implemented using CLISP%_ expressions.

23.9 Declarations

Declarations are used to affect the choice of INTERLISP function used as the translation of a particular operator. For example, A+B can be translated as either (IPLUS A B), (FPLUS A B), or (PLUS A B), depending on the declaration in effect. Similarly X:1←Y can mean (RPLACA X Y), (FRPLACA X Y), or (/RPLACA X Y), and <!!A B> either (NCONC1 A B) or (/NCONC1 A B). The table below gives the declarations available in CLISP, and the INTERLISP functions they indicate. *The choice of function on all CLISP transformations are affected by these declarations, i.e. iterative statements, pattern matches, record operations, as well as infix and prefix operators.*

The user can make (change) a global declaration by calling the function CLISPDEC and giving it as its argument a list of declarations, e.g., (CLISPDEC (QUOTE (FLOATING UNDOABLE))). Changing a global declaration does not affect the speed of subsequent CLISP transformations, since all CLISP

---

[57] Note that makefile will reset prettytranflg to T, using resetvar, when called with the option NOCLISP.

transformation are table driven (i.e. property list), and global declarations are accomplished by making the appropriate internal changes to CLISP at the time of the declaration. If a function employs *local* declarations (described below), there will be a slight loss in efficiency owing to the fact that for each CLISP transformation, the declaration list must be searched for possibly relevant declarations.

Declarations are implemented in the order that they are given, so that later declarations override earlier ones. For example, the declaration FAST specifies that FRPLACA, FRPLACD, FMEMB, and FLAST be used in place of RPLACA, RPLACD, MEMB, and LAST; the declaration RPLACA specifies that RPLACA be used. Therefore, the declarations (FAST RPLACA RPLACD) will cause FMEMB, FLAST, RPLACA, and RPLACD to be used.

The initial global declaration is INTEGER and STANDARD.

Table of Declarations

Declaration	INTERLISP functions to be used
INTEGER or FIXED	IPLUS, IMINUS, IDIFFERENCE, ITIMES, IQUOTIENT, ILESSP, IGREATERP
FLOATING	FPLUS, FMINUS, FDIFFERENCE, FTIMES, FQUOTIENT, LESSP, FGTP
MIXED	PLUS, MINUS, DIFFERENCE, TIMES, QUOTIENT, LESSP, GREATERP
FAST	FRPLACA, FRPLACD, FMEMB, FLAST, FASSOC
UNDOABLE	/RPLACA, /RPLACD, /NCONC, /NCONC1, /MAPCONC, /MAPCON
STANDARD	RPLACA, RPLACD, MEMB, LAST, ASSOC, NCONC, NCONC1, MAPCONC, MAPCON
RPLACA, RPLACD, /RPLACA, ...	corresponding function

Local Declarations

The user can also make declarations affecting a selected function or functions by inserting an expression of the form (CLISP: . declarations) immediately following the argument list, i.e., as CADDR of the definition. Such local declarations take precedence over global declarations. Declarations affecting selected variables can be indicated by lists, where the first element is the name of a variable, and the rest of the list the declarations for that variable. For example, (CLISP: FLOATING (X INTEGER)) specifies that in this function integer arithmetic be used for computations involving X, and floating arithmetic for all other computations.[58] The user can also make local record declarations by inserting a record declaration, e.g. (RECORD --), (ARRAYRECORD --), etc., in the local declaration list. Local record declarations override global record declarations for the function in which they appear. Local declarations can also be used to override the global setting of certain DWIM/CLISP parameters effective only for transformations within that function, by including in the local declaration an expression of the form (variable = value), e.g. (PATVARDEFAULT = QUOTE).

The CLISP: expression is converted to a comment of a special form recognized by CLISP. Whenever a CLISP transformation that is affected by declarations is about to be performed in a function, this comment will be searched for a relevant declaration, and if one is found, the corresponding function will be used. Otherwise, if none are found, the global declaration(s) currently in effect will be used.

---

[58] 'involving' means where the variable itself is an operand. For example, with the declaration (FLOATING (X INTEGER)) in effect, (FOO X)+(FIE X) would translate to FPLUS, i.e., use floating arithmetic, even though X appears somewhere inside of the operands, whereas X+(FIE X) would translate to IPLUS. If there are declarations involving *both* operands, e.g. X+Y, with (X FLOATING) (Y INTEGER), whichever appears first in the declaration list will be used.

Local declarations are effective in the order that they are given, so that later declarations can be used to override earlier ones, e.g. (CLISP: FAST RPLACA RPLACD) specifies that FMEMB, FLAST, RPLACA, and RPLACD be used. An exception to this is that declarations for specific variables take precedence of general, function-wide declarations, regardless of the order of appearance, as in (CLISP: (X INTEGER) FLOATING).

Clispify also checks the declarations in effect before selecting an infix operator to ensure that the corresponding CLISP construct would in fact translate back to this form. For example, if a FLOATING declaration is in effect, clispify will convert (FPLUS X Y) to X+Y, but leave (IPLUS X Y) as is. Note that if (FPLUS X Y) is CLISPIFYed while a FLOATING declaration is under effect, and then the declaration is changed to INTEGER, when X+Y is translated back to INTERLISP, it will become (IPLUS X Y).

23.10 The Pattern Match Compiler[69]

CLISP contains a fairly general pattern match facility. The purpose of this pattern match facility is to make more convenient the specifying of certain tests that would otherwise be clumsy to write (and not as intelligible), by allowing the user to give instead a pattern which the datum is supposed to match. Essentially, the user writes "Does the (expression) X look like (the pattern) P?" For example, X:(& 'A -- 'B) asks whether the second element of X is an A, and the last element a B. The implementation of the matching is performed by computing (once) the equivalent INTERLISP expression which will perform the indicated operation, and substituting this for the pattern, and *not* by invoking each time a general purpose capability such as that found in FLIP

---

[69] The pattern match compiler was written by L. M. Masinter.

or PLANNER. For example, the translation of X:(& 'A -- 'B) is: (AND (EQ (CADR X) (QUOTE A)) (EQ (CAR (LAST X)) (QUOTE B))). Thus the CLISP pattern match facility is really a Pattern Compiler, and the emphasis in its design and implementation has been more on the efficiency of object code than on generality and sophistication of its matching capabilities. The goal was to provide a facility that could and would be used even where efficiency was paramount, e.g. in inner loops. As a result, the CLISP pattern match facility does not contain (yet) some of the more esoteric features of other pattern match languages, such as repeated patterns, disjunctive and conjunctive patterns, recursion, etc. However, the user can be confident that what facilities it does provide will result in INTERLISP expressions comparable to those he would generate by hand.[60]

The syntax for pattern match expressions is form:pattern, where pattern is a list as described below. As with iterative statements, the translation of patterns, i.e., the corresponding INTERLISP expressions, are stored in clisparray, a hash array, as described on page 23.30. The original expression, form:pattern, is replaced by an expression of the form (MATCH form WITH pattern). CLISP also recognizes expressions input in this form.

If form appears more than once in the translation, and it is not either a variable, or an expression that is easy to (re)compute, such as (CAR Y), (CDDR Z), etc., a dummy variable will be generated and bound to the value of form so that form is not evaluated a multiple number of times. For example, the translation of (FOO X):($ 'A $) is simply (MEMB (QUOTE A) (FOO X)), while the translation of (FOO X):('A 'B --) is:

---

60 Wherever possible, already existing INTERLISP functions are used in the translation, e.g., the translation of ($ 'A $) uses MEMB, ($ ('A $) $) uses ASSOC, etc.

```
[PROG ($$2) (RETURN
     (AND (EQ (CAR (SETQ $$2 (FOO X)))
              (QUOTE A))
          (EQ (CADR $$2) (QUOTE B].
```

In the interests of efficiency, the pattern match compiler assumes that all lists end in NIL, i.e. there are no LISTP checks inserted in the translation to check tails. For example, the translation of X:('A & --) is (AND (EQ (CAR X) (QUOTE A)) (CDR X)), which will match with (A B) as well as (A . B). Similarly, the pattern match compiler does not insert LISTP checks on elements, e.g. X:(('A --) --) translates simply as (EQ (CAAR X) (QUOTE A)), and X:(($1 $1 --) --) as (CDDAR X).[61] Note that the user can explicitly insert LISTP checks himself by using @, as described on page 23.40, e.g. X:(($1 $1 --)@LISTP --) translates as (AND (LISTP (CAR X)) (CDDAR X)).

Pattern Elements

A pattern consists of a list of pattern elements. Each pattern element is said to match either an element of a data structure or a segment. (cf. the editor's pattern matcher, "--" matches any arbitrary segment of a list, while & or a subpattern match only one element of a list.) Those patterns which may match a segment of a list are called SEGMENT patterns; those that match a single element are called ELEMENT patterns.

---

61 The insertion of LISTP checks for *elements* is controlled by the variable patlistpcheck. When patlistpcheck is T, LISTP checks are inserted, e.g. X:(('A --) --) translates as:

    (AND (LISTP X) (LISTP (CAR X)) (EQ (CAAR X) (QUOTE A))).

patlistpcheck is initially NIL. Its value can be changed within a particular function by using a local declaration, as described on page 23.35.

Element Patterns

There are several types of element patterns, best given by their syntax:

PATTERN	MEANING
$1, or &	matches an arbitrary element of a list
'expression	matches only an element which is equal to the given expression e.g., 'A,[62] '(A B).
=form	matches only an element which is equal to the value of form, e.g., =X, =(REVERSE Y).
==form	same as =, but uses an eq check instead of equal.
atom	treatment depends on setting of patvardefault. If patvardefault is ' or QUOTE, same as 'atom. If patvardefault is = or EQUAL, same as =atom. If patvardefault is == or EQ, same as ==atom. If patvardefault is ← or SETQ, same as atom←&. patvardefault is initially =.[63]

Note: numbers and strings are always interpreted as though patvardefault were =, regardless of its setting. Eq, memb, and assoc are used for comparisons involving small integers.

---

62 eq, memb, and assoc are automatically used in the translation when the quoted expression is atomic, otherwise equal, member, and sassoc.

63 patvardefault can be changed within a particular function by using a local declaration, as described on page 23.35.

(pattern$_1$ ... pattern$_n$) n ≥ 1 matches a list which matches the given patterns, e.g., (& &), (-- 'A).

element-pattern@function-object matches an element if the element-pattern matches it, and the function-object (name of a function or a LAMBDA expression) applied to that element returns non-NIL, e.g. &@NUMBERP matches a number, ('A --)@FOO matches a list whose first element is A, and for which FOO applied to that list is non-NIL.[64]

* matches any arbitrary element. If the entire match succeeds, the element which matched the * will be returned as the value of the match.

Note: normally, the pattern match compiler constructs an expression whose value is guaranteed to be non-NIL if the match succeeds and NIL if it fails. However, if a * appears in the pattern, the expression generated will either return NIL if the match fails, or *whatever matched the ** even though that may be NIL. For example, X:('A * --) translates as (AND (EQ (CAR X) (QUOTE A)) (CDR X) (CADR X)).

~element-pattern matches an element if the element is *not* matched by element-pattern, e.g. ~'A, ~=X, ~(-- 'A --).

---

[64] For 'simple' tests, the function-object is applied before a match is attempted with the pattern, e.g. ((-- 'A --)@LISTP --) translates as (AND (LISTP (CAR X)) (MEMB (QUOTE A) (CAR X))), not the other way around.

Segment Patterns

$, or -- matches any segment of a list (including one of zero length).

The difference between $ and -- is in the type of search they generate. For example, X:($ 'A 'B $) translates as (EQ (CADR (MEMB (QUOTE A) X)) (QUOTE B)), whereas X:(-- 'A 'B $) translates as: [SOME X (FUNCTION (LAMBDA ($$2 $$1) (AND (EQ $$2 (QUOTE A)) (EQ (CADR $$1) (QUOTE B]. Thus, a paraphrase of ($ 'A 'B $) would be "Is the element following the *first* A a B?", whereas a paraphrase of (-- 'A 'B $) would be "Is there *any* A immediately followed by a B?" Note that the pattern employing $ will result in a more efficient search than that employing --. However, ($ 'A 'B $) will not match with (X Y Z A M N O A B C), but (-- 'A 'B $) will.

Essentially, once a pattern following a $ matches, the $ never resumes searching, whereas -- produces a translation that will always continue searching until there is no possibility of success. However, if the pattern match compiler can deduce from the pattern that continuing a search after a particular failure cannot possibly succeed, then the translations for both -- and $ will be the same. For example, both X:($ 'A $3 $) and (-- 'A $3 --) translate as (CDDDR (MEMB (QUOTE A) X)), because if there are not three elements following the first A, there certainly will not be three elements following subsequent A's, so there is no reason to continue searching, even for --. Similarly, ($ 'A $ 'B $) and (-- 'A -- 'B --) are equivalent.

$2, $3, etc. matches a segment of the given length. Note that $1 is not a segment pattern.

!element-pattern matches any segment which the given element pattern would match as a list. For example, if the value of FOO is

(A B C) !=FOO will match the segment ... A B C ... etc. Note that !* is permissible and means Value-of-match←$, e.g. X:($ 'A !*) translates to (CDR (MEMB (QUOTE A) X)).

Note: since ! appearing in front of the last pattern specifies a match with some *tail* of the given expression, it also makes sense in this case for a ! to appear in front of a pattern that can only match with an atom, e.g., ($2 !'A) means match if cddr of the expression is the atom A. Similarly, X:($! 'A) translates to (EQ (CDR (LAST X)) (QUOTE A)).

!atom — treatment depends on setting of patvardefault. If patvardefault is ' or QUOTE, same as !'atom (see above discussion). If patvardefault is = or EQUAL, same as !=atom. If patvardefault is == or EQ, same as !==atom. If patvardefault is ← or SETQ, same as atom←$.

The atom '.' is treated *exactly* like ![65] In addition, if a pattern ends in an atom, the '.' is first changed to !, e.g., ($1 . A) and ($1 ! A) are equivalent, even though the atom '.' does not explicitly appear in the pattern.

Segment-pattern@function-object — matches a segment if the segment-pattern matches it, and the function object applied to the corresponding segment (as a list) returns non-NIL, e.g.

---

[65] With one exception, namely '.' preceding an assignment does not have the special interpretation that ! has preceding an assignment (see page 23.43). For example, X:('A . FOO←'B) translates as: (AND (EQ (CAR X) (QUOTE A)) (EQ (CDR X) (QUOTE B)) (SETQ FOO (CDR X))), but X:('A ! FOO←'B) translates as:

```
(AND (EQ (CAR X) (QUOTE A))
     (NULL (CDDR X))
     (EQ (CADR X) (QUOTE B))
     (SETQ FOO (CDR X))).
```

(\$@CDDR 'D \$) matches (A B C D E) but not (A B D E), since CDDR of (A B) is NIL.

Note: an @ pattern applied to a segment will require *computing* the corresponding structure (with ldiff) each time the predicate is applied (except when the segment in question is a tail of the list being matched).

Assignments

Any pattern element may be preceded by a variable and a '←', meaning if the match succeeds (i.e., everything matches), the variable given is to be set to *what matches* that pattern element. For example, if X = (A B C D E), X:(\$2 Y←\$3) will set Y to (C D E). *Assignments are not performed until the entire match has succeeded.* Thus, assignments cannot be used to specify a search for an element found earlier in the match, e.g. X:(Y←\$1 =Y --)[66] will *not* match with (A A B C ...).[67] This type of match is achieved by using place-markers, described below.

If the variable is preceded by a !, the assignment is to the *tail* of the list as of that point in the pattern, i.e. that portion of the list matched by the remainder of the pattern. For example, if X is (A B C D E), X:(\$!Y←'C 'D \$) sets Y to (C D E), i.e. cddr of X. In other words, when ! precedes an assignment, it acts as a modifier to the ←, and has no effect whatsoever on the pattern itself, e.g. X:('A 'B) and X:('A !FOO←'B) match identically, and in the latter case, FOO will be set to CDR of X.

---

[66] The translation of this pattern is:
```
(COND ((AND (CDR X) (EQUAL (CADR X) Y))
        (SETQ Y (CAR X))
        T)).
```
The AND is because if Y is NIL, the pattern should match with (A NIL), but not with just (A). The T is because (CAR X) might be NIL.

[67] unless, of course, the value of Y was A before the match started.

Note: *←pattern-element and !*←pattern-element are acceptable, e.g. X:($ 'A *←('B --) --) translates as:

```
[PROG ($$2) (RETURN
     (AND (EQ (CAADR (SETQ $$2 (MEMB (QUOTE A) X)))
              (QUOTE B))
          (CADR $$2]
```

Place-markers

Variables of the form #n, n a number, are called place-markers, and are interpreted specially by the pattern match compiler. Place-markers are used in a pattern to mark or refer to a particular pattern element. Functionally, they are used like ordinary variables, i.e. they can be assigned values, or used freely in forms appearing in the pattern, e.g. X:(#1←$1 =(ADD1 #1)) will match the list (2 3). However, they are not really variables in the sense that they are not bound, nor can a function called from within the pattern expect to be able to obtain their values. For convenience, regardless of the setting of patvardefault, the first appearance of a defaulted place-marker is interpreted as though patvardefault were ←. Thus the above pattern could have been written as X:(#1 =(ADD1 #1)). Subsequent appearances of a place-marker are interpreted as though patvardefault were =. For example, X:(#1 #1 --) is equivalent to X:(#1←$1 =#1 --), and translates as (AND (CDR X) (EQUAL (CAR X) (CADR X)).[68]

Replacements

Any pattern element may be *followed* by a '←' and a form, meaning if the match succeeds, the part of the data that matched is to be *replaced* (e.g., with

---

[68] Just (EQUAL (CAR X) (CADR X)) would incorrectly match with (NIL).

RPLACA or RPLACD)[69] with the value of <form>. For example, if X =(A B C D E), X:($ 'C $1←Y $1) will replace the fourth element of X with the value of Y. As with assignments, *replacements are not performed until after it is determined that the entire match will be successful.*

Replacements involving segments splice the corresponding structure into the list being matched, e.g. if X is (A B C D E F) and FOO is (1 2 3), after the pattern ('A $←FOO 'D $) is matched with X, X will be (A 1 2 3 D E F), and FOO will be eq to CDR of x, i.e. (1 2 3 D E F).

*　　　　*　　　　*

Note that ($ FOO←FIE $) is ambiguous, since it is not clear whether FOO or FIE is the pattern element, i.e. whether ← specifies assignment or replacement. For example, if patvardefault is =, this pattern can be interpreted as ($ FOO←=FIE $), meaning search for the value of FIE, and if found set FOO to it, or ($ =FOO←FIE $) meaning search for the value of FOO, and if found, store the value of FIE into the corresponding position. In such cases, the user should disambiguate by not using the patvardefault option, i.e. by specifying ' or =.

---

69 The user can indicate he wants /rplaca and /rplacd used, or frplaca and frplacd, by means of declarations. The initial default is for rplaca and rplacd.

Reconstruction

The user can specify a value for a pattern match operation other than what is returned by the match by writing after the pattern => followed by another form, e.g. X:(FOO←$ 'A --) => (REVERSE FOO),[70] which translates as:

```
[PROG ($$2) (RETURN
     (COND ((SETQ $$2 (MEMB (QUOTE A) X))
            (SETQ FOO (LDIFF X $2))
            (REVERSE FOO].
```

Place-markers in the pattern can be referred to from within form, e.g. the above could also have been written as X:(!#1 'A --)=>(REVERSE #1). If -> is used in place of =>, the expression being matched is also *physically changed* to the value of form. For example, X:(#1 'A !#2) -> (CONS #1 #2) would remove the second element from X, if it were equal to A.

In general, form1:pattern->form2 is translated so as to compute form2 if the match is successful, and then smash its value into the first node of form1. However, whenever possible, the translation does not actually require form2 to be computed in its entirety, but instead the pattern match compiler uses form2 as an indication of what should be done to form1. For example, X:(#1 'A !#2) -> (CONS #1 #2) translates as:
(AND (EQ (CADR X) (QUOTE A)) (RPLACD X (CDDR X))).

---

70 The original CLISP is replaced by an expression of the form (MATCH form1 WITH pattern => form2). CLISP also recognizes expressions input in this form.

Examples

X:(-- 'A --)

-- matches any arbitrary segment. 'A matches only an A, and the 2nd -- again matches an arbitrary segment; thus this translates to (MEMB (QUOTE A) X).

X:(-- 'A)

Again, -- matches an arbitrary segment; however, since there is no -- after the 'A, A must be the last element of X. Thus this translates to: (EQ (CAR (LAST X)) (QUOTE A)).

X:('A 'B -- 'C $3 --)

CAR of X must be A, and CADR must be B, and there must be at least three elements after the first C, so the translation is:

```
(AND (EQ (CAR X) (QUOTE A))
     (EQ (CADR X) (QUOTE B))
     (CDDDR (MEMB (QUOTE C) (CDDR X))))
```

X:(('A 'B) 'C Y←$1 $)

Since ('A 'B) does not end in $ or --, (CDDAR X) must be NIL.

```
(COND
  ((AND (EQ (CAAR X) (QUOTE A))
        (EQ (CADAR X) (QUOTE B))
        (NULL (CDDAR X))
        (EQ (CADR X) (QUOTE C))
        (CDDR X))
    (SETQ Y (CADDR X))
    T))
```

X:(#1 'A $ 'B 'C #1 $)

#1 is implicitly assigned to the first element in the list. The $ searches for the first B following A. This B must be followed by a C, and the C by an expression equal to the first element.

```
[PROG ($$2) (RETURN
      (AND (EQ (CADR X) (QUOTE A))
           (EQ [CADR (SETQ $$2 (MEMB (QUOTE B) (CDDR X]
               (QUOTE C))
           (EQUAL (CADDR $$2) (CAR X]
```

X:(#1 'A -- 'B 'C #1 $) Similar to the pattern above, except that -- specifies a search for *any* B followed by a C followed by the first element, so the translation is:

```
[AND (EQ (CADR X) (QUOTE A))
     (SOME (CDDR X) (FUNCTION (LAMBDA ($$2 $$1)
                    (AND (EQ $$2 (QUOTE B))
                         (EQ (CADR $$1) (QUOTE C))
                         (EQUAL (CADDR $$1) (CAR X]
```

This concludes the description of the pattern match compiler.

23.11 The Record Package[71]

The advantages of "data-less" or data-structure-independent programming have long been known: more readable code, fewer bugs, the ability to change the data structure without having to make major modifications to the program, etc. The record package in CLISP both encourages and facilitates this good programming practice by providing a uniform syntax for accessing and storing data into many different types of data structures, e.g. those employing arrays, list structures, atom property lists, hash links, etc., or any combination thereof, as well as removing from the user the task of writing the various access and storage routines themselves. The user declares (once) the data structure(s) used by his programs, and thereafter indicates the manipulations of the data in a data-structure-independent manner. The record package automatically computes from the declaration(s) the corresponding INTERLISP expressions necessary to accomplish the indicated access/storage operations. The user can change his data structure simply by changing the corresponding declaration(s), and his program automatically (re)adjusts itself to the new conventions.

---

71 The record package was written by L. M. Masinter.

The user informs the record package about the format of his data structure by making a record declaration. A record declaration defines a record, i.e. a data structure. (Note that the record itself is an abstraction that exists only in the user's head.) The record declaration is essentially a template which describes the record, associating names with its various parts or *fields*. For example, the record declaration (RECORD MSG (ID (FROM TO) . TEXT)) describes a data structure called MSG, which contains four fields: ID, FROM, TO, and TEXT. The user can then reference these fields by name, either to retrieve their contents, or to store new data into them, by using the : operator followed by the field name. For example, for the above record declaration, X:FROM would be equivalent (and translate) to (CAADR X), and Y:TO←Z to (RPLACA (CDADR Y) Z).[72] The fields of a record can be further broken down into subfields by additional declarations within the record, e.g. (RECORD MSG (ID (FROM TO) . TEXT) (RECORD TEXT (HEADER TXT))) would permit the user to refer to TEXT, or to its subfields HEADER and TXT.

Note that what the record declaration is really doing is specifying the *data-paths* of the structure, and thereby specifying how the corresponding access/storage operations are to be carried out. For example, (RECORD MSG (ID (FROM TO) . TEXT) (RECORD TEXT (HEADER TXT))) says the HEADER of a MSG is to be found as the first element of its TEXT, which is the second tail of the MSG itself. Hence, X:HEADER←string is achieved by performing (RPLACA (CDDR X) string).

Note also that when the user writes X:HEADER, he is implicitly saying the X is an instance of the record MSG, or at least is to be treated as such for this particular operation. In other words, the interpretation of X:FORM *never depends on the value of X*. The record package (currently) does not provide any

---

[72] or /RPLACA or FRPLACA, depending on the CLISP declaration in effect.

facility which uses *run-time* checks to determine data paths, nor is there any error checking other than that provided by INTERLISP itself. For example, if X happened to be an array, X:HEADER would still compute (CAADR X).[73]

RECORD (used to specify elements and tails of a list structure) is just one of several record-types currently implemented. For example, the user can specify 'optional' fields, i.e. property list format, by using the record type PROPRECORD, or fields to be associated with parts of the data structure via hash links, by using the record-type HASHRECORD, or even specify the access definitions in the record declaration himself, by using the record-type ACCESSFN. These are described in detail below.

The record package also provides a facility for *creating* new data structures using a record declaration as a guide or template. Initial values for the various fields can be specified in the CREATE expression, or defaulted to values specified in the record declaration itself. Alternatively, CREATE can be instructed to use an existing datum as a model, i.e. to obtain the field values for the new datum from the corresponding fields of the existing datum, or even to actually use (cannibalize) the structure of the existing datum itself.

As with all DWIM/CLISP facilities, the record package contains many do-what-I-mean features, spelling correction on field names, record types, etc. In addition, the record package includes a RECORDS prettydef macro for dumping record declarations, as well as the appropriate modifications to the file package (Section 14), so that files? and cleanup will inform the user about records that need to be dumped.

---

[73] However, it is possible to make the interpretation of X:HEADER differ from that of Y:HEADER (regardless of the values of X and Y), by using local record declarations, as described on page 23.35. Note that this distinction depends on a *translation*-time check, not run-time.

Record Declarations

A record declaration is an expression of the form

(record-type record-name fields . {defaults and/or subfields})

This expression is evaluated to effect the corresponding declaration.[74]

1. record-type specifies the "type" of data being described by the record declaration, and thereby implicitly specifies the data paths, i.e. how the corresponding access/storage operations are performed. record-type currently is either RECORD, TYPERECORD, ARRAYRECORD, ATOMRECORD, PROPRECORD, HASHRECORD, or ACCESSFN.[75] RECORD and TYPERECORD are used to describe list structures, ARRAYRECORD to describe arrays, ATOMRECORD to describe (the property list of) atoms, and PROPRECORD to describe lists that use property list format. HASHRECORD can be used with any type of data: since it simply specifies the data path to be a hash-link. ACCESSFN is also type-less; the user specifies the data-path(s) in the record declaration itself, as described below.

2. record-name is a literal atom used to identify the record declaration for dumping to files via the RECORDS prettydef macro, and for creating instances of the record via CREATE. For most top-level declarations, record-name is optional, e.g. (RECORD (ID (FROM TO) . TEXT)) is perfectly acceptable.[76]

---

[74] Local record declarations are performed by including an expression of this form in the CLISP declaration for that function (page 23.35), rather than evaluating the expression itself.

[75] When user-defined data types are introduced to INTERLISP, a corresponding record-type will be added to the record package.

[76] If record-name is omitted, it simply means that the user cannot specify the record by name, e.g. when calling CREATE, or when using the RECORDS prettydef command.

For TYPERECORD, record-name is obligatory and is used as an indicator in CAR of the datum to signify what "type" of record it is. CREATE will insert an extra field containing record-name at the beginning of the structure, and the translation of the access and storage functions will take this extra field into account.[77]

For subfield declarations, record-name is also obligatory, and specifies the parent field that is being elaborated, as described below.

3. fields describes the structure of the record. Its exact interpretation varies with the record-type:

    For RECORD, fields is a list whose non-NIL literal atoms are taken as field-names to be associated with the corresponding elements and tails of a list structure. NIL can be used as a place marker to fill an unnamed field, e.g. (A NIL B) describes a three element list, with B corresponding to the third element.

    For TYPERECORD, fields has the same meaning as for RECORD. However, since CAR of the datum contains an indicator signifying its "type," the translation of the access/storage functions differ from those of RECORD. For example, for (TYPERECORD MSG (ID (FROM TO) . TEXT)), X:FROM translates as (CAADDR X), not (CAADR X).

    For ATOMRECORD, fields is a list of property names, e.g. (ATOMRECORD (EXPR CODE MACRO BLKLIBRARYDEF)). Accessing will be performed with getp, storing with put.

---

[77] Note: this type-field is not used by the record package. It is provided for the user's own applications.

For PROPRECORD, fields is also a list of property names. Accessing is performed with get, storing with putl.[78] For example, (RECORD ENTRY (INPUT VALUE ID . PROPS) (PROPRECORD PROPS (HISTORY LISPXPRINT SIDE GROUP ERROR))) could be used to describe an entry on the history list (see Section 22).[79]

For HASHRECORD (or HASHLINK), fields is usually just field-name, i.e. an atom, and is the name by which the corresponding hash-value is referred to. For example, for (RECORD (A B . C) (HASHRECORD B FOO)), X:FOO translates as (GETHASH (CADR X)). If field-name is a list, it is interpreted as (field-name arrayname arraysize). In this case, arrayname indicates the hash-array to be used. For example, (HASHRECORD (CLISP CLISPARRAY)) would permit the user to obtain the CLISP translation of X by simply writing X:CLISP. arraysize is used for initializing the hash array: if arrayname has not been initialized at the time of the declaration, it will be set to (HARRAY (OR arraysize 100)).

For ARRAYRECORD, fields is a list of field-names that are associated with the corresponding elements of the array. NIL can be used as a place marker for an unnamed field (element). Positive integers can be used as abbreviation for the corresponding number of NILs. For example, (ARRAYRECORD (ORG DEST NIL ID 3 TEXT)) describes an eight

---

78 A new function (part of the record package), similar to put, which takes a list as its first argument, searches the list looking for an occurrence of the given property name (its second argument). If found, it replaces the next element with the new property value (its third argument), otherwise adds the property name and property value to the list.

79 Note that (ATOMRECORD (FOO FIE)) is equivalent to (RECORD (VALUE . PROPS) (PROPRECORD PROPS (FOO FIE))), the difference being in the translations. In the first case, X:FIE translates as (GETP X (QUOTE FIE)),, in the second case, as (GET (CDR X) (QUOTE FIE)). Note also that in the first case, if X is not a literal atom, INTERLISP (i.e. getp) will generate an error.

element array, with ORG corresponding to the first element, ID to the fourth, and TEXT to the eighth.

For ACCESSFN (or ACCESSFNS), fields is a list of the form (field-name accessdefinition setdefinition), or a list of elements of this form. accessdefinition is a function of one argument, the datum, and will be used for accessing. setdefinition is a function of two arguments, the datum and the new value, and is used for storing.[80] For example, (HASHRECORD FOO) and (ACCESSFN (FOO GETHASH PUTHASH)) are equivalent: in both cases, X:FOO translates as (GETHASH FOO). Similarly, (ACCESSFN (DEF GETD PUTD)) would permit defining functions by writing fn:DEF←definition.[81]

4. {defaults and/or subfields} is optional. It may contain expressions of the form:

   (1) field-name ← form - specifies the default value for field-name. Used by CREATE.

   (2) DEFAULT ← form - specifies default value for every field not given a specific default via (1).

   (3) a subfield declaration - i.e. a record declaration of any of the above types. For subfield declarations, record-name is obligatory. Instead of identifying the declaration as with the case of top level declarations, record-name identifies the parent field or record that

---

[80] Currently, an error is generated if CREATE is called with a record declaration containing an ACCESSFNS record-type.

[81] [ACCESSFN (DEF GETD (LAMBDA (FN DEF) (DEFINE (LIST (LIST FN DEF] would be preferable to using putd.

is being described by the subfield declaration. It must be either the record-name of the immediately superior declaration, or one of its field-names (or else an error is generated).

Subfields can be nested to an arbitrary depth.

Note that in some cases, it makes sense for a given field to have more than one subfield declaration. For example, in (RECORD (A . B) (PROPRECORD B (FOO FIE FUM)) (HASHRECORD B C)), B is elaborated by both a PROPRECORD and HASHRECORD. Similarly, (RECORD (A B) (RECORD A (C D)) (RECORD A (FOO FIE))) is also acceptable, and essentially "overlays" (FOO FIE) and (C D), i.e. X:FOO and X:C would be equivalent. In such cases, the *first* subfield declaration is the one used by CREATE, e.g. (RECORD X (A B) (RECORD A (C D)) (RECORD A (FOO FIE FUM))) will cause (CREATE X) to construct ((NIL NIL) NIL), not ((NIL NIL NIL) NIL), as would be the case if the subfield declaration (RECORD A (C D)) were removed.

CREATE

Record operations can be applied to arbitrary structures, i.e. structures created directly by user programs can be manipulated in a data-independent manner using record declarations. However, to be completely data-independent, new data should be created using the same declarations that define its data paths. This can be done by means of an expression of the form (CREATE record-name . {assignments}).[82] {assignments} is optional and may

---

[82] CREATE is not defined as a function. Instead, DWIM calls the appropriate function in the record package giving it the entire CREATE expression as an argument. The translation of the CREATE expression, i.e. the INTERLISP form which is evaluated to construct the datum, is then stored elsewhere, as with iterative statements and pattern matches.

contain expressions of the following form:

(1) field-name ← form — specifies initial value for field-name.

(2) USING form — specifies that for all fields not given a value by (1), the value of the corresponding field in form is to be used.

(3) COPYING form — like USING except the corresponding values are copied (copy).

(4) REUSING form — like USING, except that wherever possible, the corresponding *structure* in form is used (similar to operation of subpair and sublis).

For example, following (RECORD FOO (A B C)),
(CREATE FOO A←T USING X) translates as (LIST T (CADR X) (CADDR X)),
(CREATE FOO A←T COPYING X)) as (LIST T (COPY (CADR X)) (COPY (CADDR X))), and
(CREATE FOO A←T REUSING X) as (CONS T (CDR X)).

A CREATE expression translates into an appropriate INTERLISP form using cons, list, put, putl, puthash, seta, etc., that creates the new datum with the various fields initialized to the appropriate values. If values are neither explicitly specified, nor implicitly specified via USING or COPYING, the

DEFAULT value in the declaration is used, if any,[83] otherwise NIL.[84]

Implementation

Record operations are implemented by replacing expressions of the form X:FOO by (FETCH FOO OF X), and X:FOO←Y by (REPLACE FOO OF X WITH Y),[85] and then storing the translation elsewhere, usually in a hash array, as described on page 23.30. Translations of CREATE expressions are also stored elswhere.

The translation of each record operation is computed using information retrieved from the property list of the field name, under the property CLISPRECORDFIELD. Thus, (global) field names must be unique, i.e. cannot be the same as the name of any other field in any other record. Records can also be declared local to a particular function by using a CLISP declaration, as described on page 23.35. Local record declarations override global ones, and a local record can have a field name the same as that of a local record of another function, or the same as a field name of a global record.

For both global and local records, the translation is computed using all CLISP declarations in effect as described on page 23.33, e.g. if the declaration UNDOABLE in in effect, /RPLACA, /RPLACD, /PUTHASH, etc. will be used.

---

[83] For RECORD and TYPERECORD declarations with non-NIL defaults, all elements and named tails will be initialized; unnamed tails will *not* be initialized. For example, (RECORD FOO (A NIL B) DEFAULT←T) will cause (CREATE FOO) to construct (T T T) not (T T T . T). Of course, (RECORD FOO (A B . C) DEFAULT←T) will cause (CREATE FOO) to construct (T T . T) as expected.

[84] For PROPRECORD, initialization is only performed where necessary. For example, (RECORD FOO (A B) (PROPRECORD (C D E))) would cause (CREATE FOO) to construct (NIL NIL), not (NIL (C NIL D NIL E NIL)). (RECORD FOO (A B) (PROPRECORD (C D E) DEFAULT←T)) however, will construct (NIL (C T D T E T)).

[85] CLISP also recognizes expressions input in this form.

When the user redeclares a global record, the translations of all expressions involving that record are automatically deleted,[86] and thus will be recomputed using the new information. If the user changes a *local* record declaration, or changes some other CLISP declaration, e.g. STANDARD to FAST, and wishes the new information to affect record expressions already translated, he must make sure the corresponding translations are removed, usually either by CLISPIFYING or changing the expression by editing it.

23.12 CLISPIFY

Clispify converts INTERLISP expressions to CLISP. Note that the expression given to clispify need *not* have originally been input as CLISP, i.e., clispify can be used on functions that were written before CLISP was even implemented. Clispify is cognizant of declaration rules as well as all of the precedence rules.[87] For example, clispify will convert (IPLUS A (ITIMES B C)) into A+B*C, but (ITIMES A (IPLUS B C)) into A*(B+C).[88] Clispify converts calls to the six basic mapping functions, MAP, MAPC, MAPCAR, MAPLIST, MAPCONC, and MAPCON, into equivalent iterative statements. It also converts certain easily recognizable internal PROG loops to the corresponding i.s. For example,

---

[86] from clisparray. If the user is not using this method for storing translations, i.e. is instead using the CLISP%_ method (page 23.31), those expressions already translated will remain as they are. (There is no practical way to locate them.)

[87] clispify is table driven exactly the same as CLISP, so that if the user changes any precedence, or defines new operators, clispify "automatically" knows about it.

[88] clispify also knows how to handle expressions consisting of a mixture of INTERLISP and CLISP, e.g. (IPLUS A B*C) is converted to A+B*C, but (ITIMES A B+C) to (A*(B+C)). clispify handles such cases by first dwimifying the expression.

```
... label (COND (pred ... forms ... (GO label))) ...
```

becomes

```
... label (WHILE pred DO ... forms ...) ...
```
[89]

Clispify is not destructive to the original INTERLISP expression, i.e. clispify produces a new expression without changing the original.[90] Clispify will not convert expressions appearing as arguments to NLAMBDA functions.[91]

The value of various global parameters affect the operation of clispify:

cl:flg

The user can disable the : transformation by setting the variable cl:flg to NIL. This will cause clispify not to transform expressions such as (CADR X) to X:2. Note that clispify does *not* convert to : notation when the argument is not atomic or a simple list (a function name and one atomic argument), regardless of the setting of this flag. The initial value of cl:flg is T.

clremparsflg

Clispify will remove parentheses in certain cases from simple forms, where 'simple' means a function name and one or two atomic arguments. For example, (COND ((ATOM X) --)) will CLISPIFY to (IF ATOM X THEN --). However, if clremparsflg is set to NIL, clispify will produce (IF (ATOM X) THEN --). Note

---

[89] clispify can convert all iterative statements input in CLISP back to CLISP, regardless of how complicated the translation was, because the original CLISP is saved.

[90] The new expression may however contain some 'pieces' of the original, since clispify attempts to minimize the number of CONSes by not copying structure whenever possible.

[91] Except for those functions for which clispify contains built in information about how they process their arguments, e.g. prog, selectq, function, progn, ersetq, nlsetq, resetvar, resetform, etc.

that regardless of the setting of this flag, the expression can be input in either form. The initial value of clremparsflg is T.

clispifypackflg

clispifypackflg affects the treatment of infix operators with atomic operands. If clispifypackflg is T, clispify will pack these into single atoms, e.g., (IPLUS A (ITIMES B C)) becomes A+B*C. If clispifypackflg is NIL, no packing is done, e.g., the above becomes A_+_B_*_C. The initial value of clispifypackflg is T.

funnyatomlst

Suppose the user has variables named A, B, and A*B. If clispify were to convert (ITIMES A B) to A*B, A*B would not translate back correctly to (ITIMES A B), since it would be the name of a variable, and therefore would not cause an error. The user can prevent this from happening by adding A*B to the list funnyatomlst. Then, (ITIMES A B) would clispify to A_*_B.

Note that A*B's appearance on funnyatomlst would *not* enable DWIM/CLISP to decode A*B+C as (IPLUS A*B.C); funnyatomlst is used only by clispify. Thus, if an identifier contains a CLISP character, it should always be separated (with spaces) from other operators. For example, if X* is a variable, the user should write (SETQ X* form) in CLISP as X* ←form, not X*←form. However, in general, it is best to *avoid use of identifiers containing CLISP character operators as much as possible*.

clispifyprettyflg

If T, causes prettyprint to clispify all expressions before printing them (but not to redefine any functions). clispifyprettyflg is temporarily reset to T, using resetvar, when makefile is called with the option CLISPIFY, or when the file in question has property FILETYPE with value CLISP on its property list. clispifyprettyflg is initially NIL.

*   *   *

In addition to the above controls, disabling a CLISP operator (see <u>cldisable</u>, page 23.74) will also disable the corresponding CLISPIFY transformation. Thus, if ← is "turned off", A←B will not transform to (SETQ A B), nor vice versa.

23.13 Dwimify

<u>Dwimify</u> is effectively a preprocessor for CLISP. <u>Dwimify</u> operates by scanning an expression as though it were being interpreted, and for each form that would generate an error, calling DWIM to 'fix' it.[92] Thus the user will see the same messages, and be asked for approval in the same situations, as he would if the expression were actually run. If DWIM is unable to make a correction, no message is printed, the form is left as it was, and the analysis proceeds.

<u>Dwimify</u> knows exactly how the interpreter works. It knows the syntax of <u>progs</u>, <u>selectqs</u>, <u>lambda</u> expressions, <u>setqs</u>, et al. It knows that the argument of <u>nlambdas</u> are not evaluated. It also knows how variables are bound. In the course of its analysis of a particular expression, <u>dwimify</u> builds a list of the bound variables from the LAMBDA expressions and PROGs that it encounters. It uses this list for spelling corrections. <u>Dwimify</u> also knows not to try to 'correct' variables that are on this list since they would be bound if the expression were actually being run. However, note that <u>dwimify</u> cannot, a priori, know about variables that are used freely but would be bound in a higher function if the expression were evaluated in its normal context. Therefore, <u>dwimify</u> will try to 'correct' these variables. Similarly, <u>dwimify</u>

---

[92] Thus <u>dwimify</u> performs *all* DWIM transformations, not just CLISP transformations, i.e., it does spelling correction, fixes 8-9 errors, handles F/L, etc.

will attempt to correct forms for which car is undefined, even when the form is not in error from the user's standpoint, but the corresponding function has simply not yet been defined.

In most cases, an attempt to transform a form that is already as the user intended will have no effect (because there will be nothing to which that form could reasonably be transformed). However, in order to avoid needless calls to DWIM or to avoid possible confusion, the user can inform dwimify *not* to attempt corrections or transformations on certain functions or variables by adding them to the list nofixfnslst or nofixvarslst respectively.[93]

Dwimify and dwimifyfns (used to dwimify several functions) maintain two internal lists of those functions and variables for which corrections were unsuccessfully attempted. These lists are initialized to nofixfnslst and nofixvarslst. Once an attempt is made to fix a particular function or variable, and the attempt fails, the function or variable is added to the corresponding list, so that on subsequent occurrences (within this call to dwimify or dwimifyfns), no attempt at correction is made. For example, if FOO calls FIE several times, and FIE is undefined at the time FOO is dwimified, dwimify will not bother with FIE after the first occurrence. In other words, once dwimify "notices" a function or variable, it no longer attempts to correct it.[94] Moreover, once dwimify "notices" such functions or variables, it subsequently treats them the same as though they were actually defined or set.

Note that these internal lists are local to each call to dwimify and dwimifyfns, so that if a function containing FOOO, a misspelled call to FOO, is

---

[93] Note that the user could achieve the same effect by simply setting the corresponding variables, and giving the functions dummy definitions.

[94] Dwimify and dwimifyfns also "notice" free variables that are set in the expression being processed.

dwimified before FOO is defined or mentioned, if the function is dwimified again after FOO has been defined, the correction will be made.

Note that the user can undo selected transformations performed by dwimify, as described in section 22.

Compiling CLISP

Since the compiler does not know about CLISP, in order to compile functions containing CLISP constructs, the definitions must first be dwimified. The user can automate this process in several ways:

1) If the variable dwimifycompflg is T, the compiler will always dwimify expressions before compiling them. dwimifycompflg is initially NIL.

2) If a file has the property FILETYPE with value CLISP on its property list, tcompl, bcompl, recompile, and brecompile will operate as though dwimifycompflg is T and dwimify all expressions before compiling.[95]

3) If the function definition has a CLISP declaration (see page 23.33), including a null declaration, i.e., just (CLISP:), the definition will be automatically dwimified before compiling.

Note: tcompl, bcompl, recompile, and brecompile all scan the entire file before doing any compiling, and take note of the names of all functions that are defined in the file as well as the names of all variables that are set by adding them to nofixfnslst and nofixvarslst, respectively. Thus, if a function

---

[95] If the value of property FILETYPE is CLISP, makefile would have automatically clispified all functions before prettyprinting, as described in Section 14.

is not currently defined, but *is* defined in the file being compiled, when dwimify is called before compiling, it will not attempt to correct the function when it appears as car of a form.

Note: compileuserfn (Section 18) is defined to call dwimify on iterative statements, as well as IF-THEN statements. Thus, if the only CLISP constructs in a function appear inside of iterative statements or IF statements, the function does not have to be dwimified before compiling.

23.14 Operation

CLISP is a part of the basic INTERLISP system. Without any special preparations, the user can include CLISP constructs in programs, or type them in directly for evaluation (in eval or apply format), and when the "error" occurrs, and DWIM is called, it will destructively[96] transform the CLISP to the equivalent INTERLISP expression and evaluate the INTERLISP expression. User approval is not requested, and no message is printed.[97]

However, if a CLISP construct contains an error, an appropriate diagnostic is generated, and the form is left unchanged. For example, if the user writes (LIST X+Y*), the error diagnostic MISSING OPERAND AT X+Y* IN (LIST X+Y*) would be generated. Similarly, if the user writes (LAST+EL X), CLISP knows that ((IPLUS LAST EL) X) is not a valid INTERLISP expression, so the error diagnostic MISSING OPERATOR IN (LAST+EL X) is generated. (For example, the user might have meant to say (LAST+EL*X).) Note that if LAST+EL were the name of a defined function, CLISP would never see this form.

---

[96] CLISP transformations, like all DWIM corrections, are undoable.

[97] This entire discussion also applies to CLISP transformation initiated by calls to DWIM from dwimify.

Since the bad CLISP transformation might not be CLISP at all, for example, it might be a misspelling of a user function or variable, DWIM holds all CLISP error messages until after trying other corrections. If one of these succeeds, the CLISP message is discarded. Otherwise, if all fail, the message is printed (but no change is made).[98] For example, suppose the user types (R/PLACA X Y). CLISP generates a diagnostic, since ((IQUOTIENT R PLACA) X Y) is obviously not right. However, since R/PLACA spelling corrects to /RPLACA, this diagnostic is never printed.

If a CLISP infix construct is well formed from a syntactic standpoint, but one or both of its operands are atomic and not bound,[99] it is possible that either the operand is misspelled, e.g., the user wrote X+YY for X+Y, or that a CLISP transformation operation was not intended at all, but that the entire expression is a misspelling. For example, if the user has a variable named LAST-EL, and writes (LIST LAST-ELL). Therefore, CLISP computes, but does not actually perform, the indicated infix transformation. DWIM then continues, and if it is able to make another correction, does so, and ignores the CLISP interpretation. For example, with LAST-ELL, the transformation LAST-ELL -> LAST-EL would be found.

If no other transformation is found, and DWIM is about to interpret a construct as CLISP for which one of the operands is not bound, DWIM will ask the user whether CLISP was intended, in this case by printing

---

98 Except that CLISP error messages are not printed on type-in. For example, typing X+*Y will just produce a U.B.A. X+*Y message.

99 For the purpose of <u>dwimifying</u>,, 'not bound' means no top level value, not on list of bound variables built up by <u>dwimify</u> during its analysis of the expression, and not on <u>nofixvarslst</u>, i.e., not previously seen.

LAST-ELL TREAT AS CLISP ?[100]

The same sort of procedure is followed with 8 and 9 errors. For example, suppose the user writes FOO8*X where FOO8 is not bound. The CLISP transformation is noted, and DWIM proceeds. It next asks the user to approve FOO8*X -> FOO (*X. (For example, this would make sense if the user has (or plans to define) a function named *X.) If he refuses, the user is asked whether FOO8*X is to be treated as CLISP. Similarly, if FOO8 were the name of a variable, and the user writes FOOO8*X, he will first be asked to approve FOOO8*X -> FOOO (XX,[101] and if he refuses, then be offered the FOOO8 -> FOO8 correction.

CLISP also contains provision for correcting misspellings of infix operators (other than single characters), IF words, and i.s. operators. This is implemented in such a way that the user who does not misspell them is not penalized. For example, if the user writes IF N=0 THEN 1 ELSSE N*(FACT N-1) CLISP does *not* operate by checking each word to see if it is a misspelling of IF, THEN, ELSE, or ELSEIF, since this would seriously degrade CLISP's performance on *all* IF statements. Instead, CLISP assumes that all of the IF words are spelled correctly, and transforms the expression to (COND ((ZEROP N) 1 ELSSE N*(FACT N-1))). Later, after DWIM cannot find any other interpretation for ELSSE, and using the fact that this atom originally appeared in an IF statement, DWIM attempts spelling correction, using

---

[100] If more than one infix operator was involved in the CLISP construct, e.g., X+Y+Z, or the operation was an assignment to a variable already noticed, or treatasclispflg is T (initially NIL), the user will simply be informed of the correction. Otherwise, even if DWIM was enabled in TRUSTING mode, the user will be asked to approve the correction.

[101] The 8-9 transformation is tried before spelling correction since it is empirically more likely that an unbound atom or undefined function containing an 8 or a 9 is a parenthesis error, rather than a spelling error.

(IF THEN ELSE ELSEIF) for a spelling list. When this is successful, DWIM 'fails' all the way back to the original IF statement, changes ELSSE to ELSE, and starts over. Misspellings of AND, OR, LT, GT, etc. are handled similarly.

CLISP also contains many Do-What-I-Mean features besides spelling corrections. For example, the form (LIST +X Y) would generate a MISSING OPERATOR error. However, (LIST -X Y) makes sense, if the minus is unary, so DWIM offers this interpretation to the user. Another common error, especially for new users, is to write (LIST X*FOO(Y)) or (LIST X*FOO Y), where FOO is the name of a function, instead of (LIST X*(FOO Y)). Therefore, whenever an operand that is not bound is also the name of a function (or corrects to one), the above interpretations are offered.

23.15 CLISP Interaction with User

Syntactically and semantically well formed CLISP transformations are always performed without informing the user. Other CLISP transformations described in the previous section, e.g. misspellings of operands, infix operators, parentheses errors, unary minus - binary minus errors, all follow the same protocol as other DWIM transformations (Section 17). That is, if DWIM has been enabled in TRUSTING mode, or the transformation is in an expression typed in by the user for immediate execution, user approval is not requested, but the user is informed.[102] However, if the transformation involves a user program, and DWIM was enabled in CAUTIOUS mode, the user will be asked to approve. If he says NO, the transformation is not performed. Thus, in the previous section,

---

[102] However, in certain situations, DWIM will ask for approval even if DWIM is enabled in TRUSTING mode. For example, the user will always be asked to approve a spelling correction that might also be interpreted as a CLISP transformation, as in LAST-ELL -> LAST-EL.

phrases such as "one of these (transformations) succeeds" and "the transformation LAST-ELL -> LAST-EL would be found" etc., all mean if the user is in CAUTIOUS mode and the error is in a program, the corresponding transformation will be performed only if the user approves (or defaults by not responding). If the user says NO, the procedure followed is the same as though the transformation had not been found. For example, if A*B appears in the function FOO, and B is not bound (and no other transformations are found) the user would be asked

A*B [IN FOO] TREAT AS CLISP ? [103]

If the user approved, A*B would be transformed to (ITIMES A B), which would then cause a U.B.A. B error in the event that the program was being run (remember the entire discussion also applies to DWIMIFYing). If the user said NO, A*B would be left alone.

23.16 CLISP Internal Conventions

Note: the reader can skip this section and proceed to "Function and Variables" (page 23.71), unless he wants to add new operators, or modify the action of existing ones (other than by making declarations).

CLISP is almost entirely table driven by property lists for the corresponding infix or prefix operators. Thus it is relatively easy to add new infix or prefix operators or change old ones, simply by adding or changing selected property values.[104]

---

[103] The waiting time on such interactions is three times as long as for simple corrections, i.e., 3*dwimwait.

[104] There is some built in information for handling minus, :, ', <, >, and ~, i.e., the user could not himself add such 'special' operators, although he can disable them.

CLISPTYPE

The property value of the property CLISPTYPE is the precedence number of the operator:[105] higher values have higher precedence, i.e. are tighter. Note that the actual value is unimportant, only the value relative to other operators. For example, CLISPTYPE for :, ↑, and * are 14, 6, and 4 respectively. Operators with the same precedence group left to right, e.g., / also has precedence 4, so A/B*C is (A/B)*C.

An operator can have a different left and right precedence by making the value of CLISPTYPE be a dotted pair of two numbers, e.g., CLISPTYPE of ← is (8 . -12). In this case, car is the left precedence, and cdr the right, i.e., car is used when comparing with operators on the left, and cdr with operators on the right. For example, A*B←C+D is parsed as A*(B←(C+D)) because the left precedence of ← is 8, which is higher than that of *, which is 4. The right precedence of ← is -12, which is lower than that of +, which is 2.

If the CLISPTYPE property for any infix operator is removed, the corresponding CLISP transformation is disabled, as well as the inverse CLISPIFY transformation.

UNARYOP

The value of property UNARYOP must be T for unary operators. The operand is always on the right, i.e., unary operators are always prefix operators.

---

[105] Unless otherwise specified, the property is stored on the property list of the operator.

BROADSCOPE — The value of property BROADSCOPE is T if the operator has lower precedence than INTERLISP forms, e.g., LT, EQUAL, AND, etc. For example, (FOO X AND Y) parses as ((FOO X) AND Y). If the BROADSCOPE property were removed from the property list of AND, (FOO X AND Y) would parse as (FOO (X AND Y)).

LISPFN — The value of the property LISPFN is the name of the function to which the infix operator translates. For example, the value of LISPFN for ↑ is EXPT, for ' QUOTE, etc. If the value of the property LISPFN is NIL, the infix operator itself is also the function e.g., AND, OR, EQUAL.

SETFN — If FOO has a SETFN property FIE, then (FOO --)←X translates to (FIE -- X). For example, if the user makes ELT be an infix operator, e.g. #, by putting appropriate CLISPTYPE and LISPFN properties on the property list of # then he can also make # followed by ← translate to SETA, e.g. X#N←Y to (SETA X N Y), by putting SETA on the property list of ELT under the property SETFN. Putting (ELT) (i.e. list[ELT])) on the property list of SETA under property SETFN will enable SETA forms to CLISPIFY back to ELT's.

CLISPINFIX — The value of this property is the CLISP infix to be used in CLISPIFYing. This property is stored on the property list of the corresponding INTERLISP function, e.g., the value of property CLISPINFIX for EXPT is ↑, for QUOTE is ' etc.

Global declarations operate by changing the corresponding LISPFN and CLISPINFIX properties.

clispchars — is a list of single character operators that can appear in the interior of an atom. Currently these are: +, -, *, /, ↑, ~, ', =, ←, :, <, and >.

clispcharray — is a bit table of the characters on clispchars used for calls to strposl (see Section 10). clispcharray is initialized by performing
(SETQ CLISPCHARRAY (MAKEBITTABLE CLISPCHARS)).

clispinfixes — is a list of infix operators used for spelling correction.

As an example, suppose the user wants to make | be an infix character operator meaning OR. He performs:

```
←(PUT (QUOTE |) (QUOTE CLISPTYPE) (GETP (QUOTE OR) (QUOTE CLISPTYPE)))
←PUT(| LISPFN OR)
←PUT(| BROADSCOPE T)
←PUT(OR CLISPINFIX |)
←SETQ(CLISPCHARS (CONS (QUOTE |) CLISPCHARS))
←SETQ(CLISPCHARRAY (MAKEBITTABLE CLISPCHARS))
```

23.17 CLISP Functions and Variables

clispflg — if set to NIL, disables all CLISP infix or prefix transformations (but does not affect IF/THEN/ELSE statements, or iterative statements).

If clispflg=TYPE-IN, CLISP transformations are performed only on expressions that are typed in for evaluation, i.e. not on user programs.

If clispflg=T, CLISP transformations are performed on all expressions.

The initial value for clispflg is T. clispifying anything will cause clispflg to be set to T.

clisparray

hash array used for storing translations. clisparray is checked by faulteval on erroneous forms before calling DWIM, and by the compiler.

nofixfnslst

list of functions that dwimify will not try to correct. See page 23.62.

nofixvarslst

list of variables that dwimify will not try to correct. See page 23.62.

nospellflg

If nospellflg is T, dwimify will not perform any spelling corrections. The initial value of nospellflg is NIL.

dwimify[x;l]

dwimifies x, i.e., performs all corrections and transformations that would be performed if x were run. If x is an atom and l is NIL, x is treated as the name of a function, and its entire definition is dwimified. Otherwise, if x is a list or l is not NIL, x is the expression to be dwimified. If l is not NIL, it is the edit push-down list leading to x, and is used for determining context, i.e., what bound variables would be in effect when x was evaluated, whether x is a form

or sequence of forms, e.g., a cond clause, etc.[106]

dwimifyfns[fns] — nlambda, nospread. Dwimifies each function on fns. If fns consists of only one element, the value of car[fns] is used, e.g., dwimifyfns[FOOFNS]. Every 30 seconds, dwimifyfns prints the name of the function it is processing, a la prettyprint.

dwimifycompflg — if T, dwimify is called before compiling an expression. See page 23.63.

clispdec[declst] — puts into effect the declarations in declst. clispdec performs spelling corrections on words not recognized as declarations. clispdec is undoable.

clispify[x;l] — clispifies x. If x is an atom and l is NIL, x is treated as the name of a function, and its definition (or EXPR property) is clispified. After clispify has finished, x is redefined (using /PUTD) with its new CLISP definition. The value of clispify is x. If x is atomic and not the name of a function, spelling correction is attempted. If this fails, an error is generated.

If x is a list, or l is not NIL, x itself is the expression to be clispified. If l is not NIL, it is the edit push-down list leading to x and is used to

---

[106] If x is an iterative statement and l is NIL, dwimify will also print the translation, i.e. what is stored in the hash array.

determine context as with dwimify, as well as to obtain the local declarations, if any. The value of clispify is the clispified version of x.

See earlier section on CLISPIFY for more details.

clispifyfns[fns] — nlambda, nospread. Calls clispify on each member of fns under errorset protection. If fns consists of only one element, the value of car[fns] is used, e.g., clispifyfns[FOOFNS]. Every 30 seconds, clispifyfns prints the name of the function it is working, a la prettyprint. Value is list of functions clispifyed.

cldisable[op] — disables op, e.g. cldisable[-] makes - be just another character. cldisable can be used on all CLISP operators, e.g. infix operators, prefix operators, iterative statement operators, etc. cldisable is undoable.

clispiftranflg — affects handling of translations of IF|THEN|ELSE statements. If T, the translations are stored elsewhere, and the (modified) CLISP retained. If NIL, the corresponding COND expression, replaces the CLISP. clispiftranflg is initially NIL. See page 23.30.

cl:flg — affects clispify's handling of forms beginning with car, cdr, ... cddddr. See page 23.59.

clremparsflg — affects clispify's removal of parentheses from "small" forms. See page 23.59.

clispifypackflg — if T, informs clispify to pack operator and atomic operands into single atoms; if NIL, no packing is done. See page 23.60.

clispifyprettyflg — if T, causes prettyprint to CLISPIFY expressions before printing them. clispifyprettyflg is (temporarily) reset to T when makefile is called with the option CLISPIFY. clispifyprettyflg is initially NIL.

prettytranflg — If T, causes prettyprint to print translations instead of CLISP expressions. This is useful for creating a file for compilation, or for exporting to a LISP system that does not have CLISP. prettytranflg is (temporarily) reset to T when makefile is called with the option NOCLISP. If prettytranflg is CLISP%_, both the CLISP and translations are printed in appropriate form. For more details, see page 23.32. prettytranflg is initially NIL.

PPT — is both a function and an edit macro for prettyprinting translations. It performs a PP after first resetting prettytranflg to T, thereby causing any translations to be printed instead of the corresponding CLISP.

CLISP: — edit macro that obtains the translation of the correct expression, if any, from clisparray, and calls edite on it.

funnyatomlst — list of identifiers containing CLISP operators. Used by clispify to avoid accidentally constructing a user identifier, e.g., (ITIMES A B) should not become A*B if A*B is the name of a PROG variable. See page 23.60.

CL — edit macro. Replaces current expression with CLISPIFYed current expression. Current expression can be an element or tail.

DW — edit macro. DWIMIFYs current expression, which can be an element (atom or list) or tail.

Both CL and DW can be called when the current expression is either an element or a tail and will work properly. Both consult the declarations in the function being edited, if any, and both are undoable.

lowercase[flg] — If flg=T, lowercase makes the necessary internal modifications so that clispify will use lower case versions of AND, OR, IF, THEN, ELSE, ELSEIF, and all i.s. operators. This produces more readable output. Note that the user can always type in *either* upper or lower case (or a combination), regardless of the action of lowercase.

If flg=NIL, clispify will use uppercase versions of AND, OR, et al. The value of lowercase is its previous 'setting'. Lowercase is undoable.

lowercase also sets model33flg to null[flg], as well as performing raise[null[flg]]. model33flg affects the operation of the spelling corrector to the extent that it says something about the layout of the keyboard (see Section 17). raise[T] is the same as the TENEX command RAISE, and informs TENEX to raise all lower case characters on input, i.e. convert them to uppercase. raise[] corresponds to NO RAISE.

Index for Section 23

Page Numbers

APPENDICES

Appendix 1

Transor

Introduction

transor is a LISP-to-LISP translator intended to help the user who has a program coded in one dialect of LISP and wishes to carry it over to another. The user loads transor along with a file of transformations. These transformations describe the differences between the two LISPs, expressed in terms of INTERLISP editor commands needed to convert the old to new, i.e. to edit forms written in the source dialect to make them suitable for the target dialect. transor then sweeps through the user's program and applies the edit transformations, producing an object file for the target system. In addition, transor produces a file of translation notes, which catalogs the major changes made in the code as well as the forms that require further attention by the user. Operationally, therefore, transor is a facility for conducting massive edits, and may be used for any purpose which that may suggest.

Since the edit transformations are fundamental to this process, let us begin with a definition and some examples. A transformation is a list of edit commands associated with a literal atom, usually a function name. transor conducts a sweep through the user's code, until it finds a form whose car is a literal atom which has a transformation. The sweep then pauses to let the editor execute the list of commands before going on. For example, suppose the order of arguments for the function tconc must be reversed for the target system. The transformation for tconc would then be: ((SW 2 3)). When the

sweep encounters the form (TCONC X (FOO)), this transformation would be retrieved and executed, converting the expression to (TCONC (FOO) X). Then the sweep would locate the next form, in this case (FOO), and any transformations for *foo* would be executed, etc.

Most instances of *tconc* would be successfully translated by this transformation. However, if there were no second argument to *tconc*, e.g. the form to be translated was (TCONC X), the command (SW 2 3) would cause an error, which *transor* would catch. The sweep would go on as before, but a note would appear in the translation listing stating that the transformation for this particular form failed to work. The user would then have to compare the form and the commands, to figure out what caused the problem. One might, however, anticipate this difficulty with a more sophisticated transformation: ((IF (## 3) ((SW 2 3)) ((-2 NIL)))), which tests for a third element and does (SW 2 3) or (-2 NIL) as appropriate. It should be obvious that the translation process is no more sophisticated than the transformations used.

This documentation is divided into two main parts. The first describes how to use *transor* assuming that the user already has a complete set of transformations. The second documents *transorset*, an interactive routine for building up such sets. *transorset* contains commands for writing and editing transformations, saving one's work on a file, testing transformations by translating sample forms, etc.

Two transformations files presently exist for translating programs into INTERLISP. <LISP>SDS940.XFORMS is for old BBN LISP (SDS 940) programs, and <LISP>LISP16.XFORMS is for Stanford AI LISP 1.6 programs. A set for LISP 1.5 is planned.

Using Transor

The first and most exasperating problem in carrying a program from one implementation to another is simply to get it to read in. For example, SRI LISP uses / exactly as INTERLISP uses %, i.e. as an escape character. The function prescan exists to help with these problems: the user uses prescan to perform an initial scan to dispose of these difficulties, rather than attempting to transor the foreign sourcefiles directly.

prescan copies a file, performing character-for-character substitutions. It is hand-coded and is much faster than either readc's or text-editors.

prescan[file;charlst] — Makes a new version of file, performing substitutions according to charlst. Each element of charlst must be a dot-pair of two character codes, (OLD . NEW).

For example, SRI files are prescan'ed with charlst = ((37 . 47) (47 . 37)), which exchanges slash (47) and percent-sign (37).

The user should also make sure that the treatment of doublequotes by the source and target systems is similar. In INTERLISP, an unmatched double-quote (unless protected by the escape character) will cause the rest of the file to read in as a string.

Finally, the lack of a STOP at the end of a file is harmless, since transor will suppress END OF FILE errors and exit normally.

Translating

transor is the top-level function of the translator itself, and takes one

argument, a file to be translated. The file is assumed to contain a sequence of forms, which are read in, translated, and output to a file called file.TRAN. The translation notes are meanwhile output to file.LSTRAN. Thus the usual sequence for bring a foreign file to INTERLISP is as follows: prescan the file; examine code and transformations, making changes to the transformations if needed; transor the file; and clean up remaining problems, guided by the notes. The user can now make a pretty file and proceed to exercise and check out his program. To export a file, it is usually best to transor it, then prescan it, and perform clean-up on the foreign system where the file can be loaded.

transor[sourcefile] — Translates sourcefile. Prettyprints translation on file.TRAN: translation listing on file.LSTRAN.

transorform[form] — Argument is a LISP form. Returns the (destructively) translated form. The translation listing is dumped to the primary output file.

transorfns[fnlst] — Argument is a list of function names whose interpreted definitions are destructively translated. Listing to primary output file.

transform and transorfns can be used to translate expressions that are already in core, whereas transor itself only works on files.

The Translation Notes

The translation notes are a catalog of changes made in the user's code, and of problems which require, or may require, further attention from the user. This catalog consists of two cross-indexed sections: an index of forms and an index of notes. The first tabulates all the notes applicable to any form, whereas

the second tabulates all the forms to which any one note applies. Forms appear in the index of forms in the order in which they were encountered, i.e. the order in which they appear on the source and output files. The index of notes shows the name of each note, the entry numbers where it was used, and its text, and is alphabetical by name. The following sample was made by translating a small test file written in SRI LISP.

LISTING FROM TRANSORING OF FILE TESTFILE.;5
DONE ON 1-NOV-71 20:10:47

INDEX OF FORMS

```
1. APPLY/EVAL at
  [DEFINEQ
     (FSET (LAMBDA &
              (PROG ...3...
                    (SETQ Z (COND
                          ((ATOM (SETQ --))
                            (COND
                              ((ATOM (SETQ Y (NLSETQ "(EVAL W)")))
                                --)
                              --))
                          --))
              --   ]
2. APPLY/EVAL at
  [DEFINEQ
     (FSET (LAMBDA &
              (PROG ...3...
                    (SETQ Z (COND
                          ((ATOM (SETQ --))
                          · (COND
                              ((ATOM (SETQ --))
                                "(EVAL (NCONS W))")
                              --))
                          --))
              --   ]
3. MACHINE-CODE at
  [DEFINEQ
     (LESS1 (LAMBDA &
              (PROG ...3...
                    (COND
                      ...2...
                      ((NOT (EQUAL (SETQ X2 "(OPENR (MAKNUM & -))"
                                          )
                                        --))
                        --))
              --    ]
4. MACHINE-CODE at
  [DEFINEQ
     (LESS1 (LAMBDA &
              (PROG ...3...
                    (COND
                      ...2...
                      ((NOT (EQUAL & (SETQ Y2
                                     "(OPENR (MAKNUM & --))")))

                        --))
                --    ]
```

INDEX OF NOTES

APPLY/EVAL at 1, 2.
TRANSOR will translate the arguments of the APPLY or EVAL expression, but the user must make sure that the run-time evaluation of the arguments returns a BBN-compatible expression.
MACHINE-CODE at 3, 4.
Expression dependent on machine-code. User must recode.

The translation notes are generated by the transformations used, and therefore reflect the judgment of their author as to what should be included. Straightforward conversions are usually made without comment; for example, the DEFPROP's in this file were quietly changed to DEFINEQ's. _transor_ found four noteworthy forms on the file, and printed an entry for each in the index of forms, consisting of an entry number, the name of the note, and a printout showing the precise location of the form. The form appears in double-quotes and is the last thing printed, except for closing parentheses and dashes. An ampersand represents one non-atomic element not shown, and two or more elements not shown are represented as ...n..., where n is the number of elements. Note that the printouts describe expressions on the output file rather than the source file; in the example, the DEFPROP's of SRI LISP have been replaced with DEFINEQ's.

Errors and Messages

transor records its progress through the source file by teletype printouts which identify each expression as it is read in. Progress within large expressions, such as a long DEFINEQ, is reported every three minutes by a printout showing the location of the sweep.

If a transformation fails, _transor_ prints a diagnostic to the teletype which identifies the faulty transformation, and resumes the sweep with the next form. The translation notes will identify the form which caused this failure, and the extent to which the form and its arguments were compromised by the error.

If the transformation for a common function fails repeatedly, the user can type control-H. When the system goes into a break, he can use _transorset_ to repair the transformation, and even test it out (see TEST command, page A1.11). He may then continue the main translation with OK.

Transorset

To use transorset, type transorset() to INTERLISP. transorset will respond with a + sign, its prompt character, and await input. The user is now in an executive loop which is like evalqt with some extra context and capabilities intended to facilitate the writing of transformations. transorset will thus progress apply and eval input, and execute history commands just as evalqt would. Edit commands, however, are interpreted as additions to the transformation on which the user is currently working. transorset always saves on a variable named currentfn the name of the last function whose transformation was altered or examined by the user. currentfn thus represents the function whose transformation is currently being worked on. Whenever edit commands are typed to the + sign, transorset will add them to the transformation for currentfn. This is the basic mechanism for writing a transformation. In addition, transorset contains commands for printing out a transformation, editing a transformation, etc., which all assume that the command applies to currentfn if no function is specified. The following example illustrates this process.

```
←TRANSORSET()
+FN TCONC                                          [1]
TCONC
+(SW 2 3)                                          [2]
+TEST (TCONC A B)                                  [3]
P
(TCONC B A)
+TEST (TCONC X)                                    [4]
TRANSLATION ERROR: FAULTY TRANSFORMATION
TRANSFORMATION: ((SW 2 3))                         [5]
OBJECT FORM:    (TCONC X)


1. TRANSFORMATION ERROR AT                         [6]
  "(TCONC X)"


(TCONC X)
+(IF (## 3) ((SW 2 3)) ((-2 NIL]                   [7]
+SHOW
TCONC
  [(SW 2 3)
   (IF (## 3)                                      [8]
       ((SW 2 3))
       ((-2 NIL]
TCONC
+ERASE                                             [9]
TCONC
+REDO IF                                           [10]
+SHOW
TCONC
  [(IF (## 3)
       ((SW 2 3))
       ((-2 NIL]
TCONC
+TDST
=TEST                                              [11]
(TCONC NIL X)
+
```

In this example, the user begins by using the FN command to set currentfn to TCONC [1]. He then adds to the (empty) transformation for tconc a command to switch the order of the arguments [2] and tests the transformation [3]. His second TEST [4] fails, causing an error diagnostic [5] and a translation note [6]. He writes a better command [7] but forgets that the original SW command is still in the way [8]. He therefore deletes the entire transformation [9] and redoes the IF [10]. This time, the TEST works [11].

Transorset Commands

The following commands for manipulating transformations are all lispxmacros which treat the rest of their input line as arguments. All are undoable.

FN — Resets currentfn to its argument, and returns the new value. In effect FN says you are done with the old function (as least for the moment) and wish to work on another. If the new function already has a transformation, the message (OLD TRANSFORMATIONS) is printed, and any editcommands typed in will be added to the end of the existing commands. FN followed by a carriage return will return the value of currentfn without changing it.

SHOW — Command to prettyprint a transformation. SHOW followed by a carriage return will show the transformation for currentfn, and return currentfn as its value. SHOW followed by one or more function names will show each one in turn, reset currentfn to the last one, and return the new value of currentfn.

EDIT — Command to edit a transformation. Similar to SHOW except that instead of prettyprinting the transformation, EDIT gives it to edite. The user can then work on the transformation until he leaves the editor with OK.

ERASE — Command to delete a transformation. Otherwise similar to SHOW.

TEST — Command for checking out transformations. TEST takes one argument, a form for translation. The translation notes, if any, are printed to the teletype, but in an abbreviated format which omits the index of notes. The value returned is the translated form. TEST saves a copy of its argument on the free variable testform, and if no argument is given, it uses testform, i.e. tries the previous test again.

DUMP — Command to save your work on a file. DUMP takes one argument, a filename. The argument is saved on the variable dumpfile, so that if no argument is provided, a new version of the previous file will be created.

The DUMP command creates files by makefile. Normally fileFNS will be unbound, but the user may set it himself; functions called from a transformation by the E command may be saved in this way. DUMP makes sure that the necessary command is included on the fileVARS to save the user's transformations. The user may add anything else to his fileVARS that he wishes. When a transformation file is loaded, all previous transformations are erased unless the variable merge is set to T.

EXIT transorset returns NIL.

The REMARK Feature

The translation notes are generated by those transformations that are actually executed via an editmacro called REMARK. REMARK takes one argument, the name of a note. When the macro is executed, it saves the appropriate information for the translation notes, and adds one entry to the index of forms. The location that is printed in the index of forms is the editor's location when the REMARK macro is executed.

To write a transformation which makes a new note, one must therefore do two things: define the note, i.e. choose a new name and associate it with the desired text; and call the new note with the REMARK macro, i.e. insert the edit command (REMARK name) in some transformation. The NOTE command, described below, is used to define a new note. The call to the note may be added to a transformation like any other edit command. Once a note is defined, it may be called from as many different transformations as desired.

The user can also specify a remark with a new text, without bothering to think of a name and perform a separate defining operation, by calling REMARK with more than one argument, e.g. (REMARK text-of-remark). This is interpreted to mean that the arguments are the text. transorset notices all such expressions as they are typed in, and handles naming automatically; a new name is generated[1] and defined with the text provided, and the expression itself is edited to be (REMARK generated-name). The following example illustrates the use of REMARK.

---

[1] The name generated is the value of currentfn suffixed with a colon, or with a number and a colon.

```
←TRANSORSET()
+NOTE GREATERP/LESSP (BBN'S GREATERP AND LESSP ONLY          [1]
TAKE TWO ARGUMENTS, WHEREAS SRI'S FUNCTIONS TAKE AN
INDEFINITE NUMBER. AT THE PLACES NOTED HERE, THE SRI CODE
USED MORE THAN TWO ARGUMENTS, AND THE USER MUST RECODE.]
GREATERP/LESSP
+FN GREATERP
GREATERP
+(IF (IGREATERP (LENGTH (##))3) NIL ((REMARK GREATERP/LESSP] [2]
+FN LESSP
LESSP
+REDO IF                                                      [3]
+SHOW
LESSP
  [(IF (IGREATERP (LENGTH (##))
                  3)
       NIL
       ((REMARK GREATERP/LESSP]
LESSP
+FN ASCII
(OLD TRANSFORMATIONS)
ASCII
+(REMARK ALTHOUGH THE SRI FUNCTION ASCII IS IDENTICAL        [4]
TO THE BBN FUNCTION CHARACTER, THE USER MUST MAKE SURE THAT
THE CHARACTER BEING CREATED SERVES THE SAME PURPOSE ON BOTH
SYSTEMS, SINCE THE CONTROL CHARACTERS ARE ALL ASSIGNED
DIFFRENTLY.]
+SHOW                                                         [5]
ASCII
  ((1 CHARACTER)
   (REMARK ASCII:))
ASCII
+NOTE ASCII:                                                  [6]
EDIT
*NTH -2
*P
... ASSIGNED DIFFRENTLY.)
*(2 DIFFERENTLY.)
OK
ASCII:
+
```

In this example, the user defines a note named GREATERP/LESSP by using the NOTE command [1], and writes transformations which call this note whenever the sweep encounters a GREATERP or LESSP with more than two arguments [2-3]. Next, the implicit naming feature is used [4] to add a REMARK command to the transformation for ASCII, which has already been partly written. The user realizes he mistyped part of the text, so he uses the SHOW command to find the name chosen for the note [5]. Then he uses the NOTE command on this name, ASCII:, to edit the note [6].

NOTE	First argument is note name and must be a literal atom. If already defined, NOTE edits the old text; otherwise it defines the name, reading the text either from the rest of the input line or from the next line. The text may be given as a line or as a list. Value is name of note.

The text is actually stored.[2] as a comment, i.e. a * and %% are added in front when the note is first defined. The text will therefore be lower-cased the first time the user DUMPs (see Section 14).

DELNOTE	Deletes a note completely (although any calls to it remain in the transformations).

Controlling the Sweep

transor's sweep searches in print-order until it finds a form for which a transformation exists. The location is marked, and the transformation is

---

[2] On the global list _usernotes_.

executed. The sweep then takes over again, beginning from the marked location, no matter where the last command of the transformation left the editor. User transformations can therefore move around freely to examine the context, without worrying about confusing the translator. However, there are many cases where the user wants his transformation to guide the sweep, usually in order to direct the processing of special forms and FEXPR's. For example, the transformation for QUOTE has only one objective: to tell the sweep to skip over the argument to QUOTE, which is (presumably) not a LISP form. NLAM is an editmacro to permit this.

NLAM — An atomic editmacro which sets a flag which causes the sweep to skip the arguments of the current form when the sweep resumes.

Special forms such as cond, prog, selectq, etc., present a more difficult problem. For example, (COND (A B)) is processed just like (FOO (A B)): i.e. after the transformation for cond finishes, the sweep will locate the "next form," (A B), retrieve the transformation for the function A, if any, and execute it. Therefore, special forms must have transformations that preempt the sweep and direct the translation themselves. The following two atomic editmacros permit such transformations to process their forms, translating or skipping over arbitrary subexpressions as desired.

DOTHIS — Translates the editor's current expression, treating it as a single form.

DOTHESE — Translates the editor's current expression, treating it as a list of forms.

For example, a transformation for setq might be (3 DOTHIS).[3] This translates the second argument to a setq without translating the first. For cond, one might write (1 (LPQ NX DOTHESE)), which locates each clause of the COND in turn, and translates it as a list of forms, instead of as a single form.

The user who is starting a completely new set of transformations must begin by writing transformations for all the special forms. To assist him in this and prevent oversights, the file <LISP>SPECIAL.XFORMS contains a set of transformations for LISP special forms, as well as some other transformations which should also be included. The user will probably have to revise these transformations substantially, since they merely perform sweep control for INTERLISP, i.e. they make no changes in the object code. They are provided chiefly as a checklist and tutorial device, since these transformations are both the first to be written and the most difficult, especially for users new to the INTERLISP editor.

* * *

When the sweep mechanism encounters a form which is not a list, or a form car of which is not an atom, it retrieves one of the following special transformations.

NLISTPCOMS — Global value is used as a transformation for any form which is not a list.

For example, if the user wished to make sure that all strings were quoted, he might set nlistpcoms to

((IF (STRINGP (##)) ((ORR ((← QUOTE))((MBD QUOTE)))) NIL)).

---

[3] Recall that a transformation is a list of edit commands. In this case, there are two commands, 3 and DOTHIS.

LAMBDACOMS — Global value is used as a transformation for any form, car of which is not an atom.

These variables are initialized by <LISP>SPECIAL.XFORMS and are saved by the DUMP command. nlistpcoms is initially NIL, making it a NOP. lambdacoms is initialized to check first for open LAMBDA expressions, processing them without translation notes unless the expression is badly formed. Any other forms with a non-atomic car are simply treated as lists of forms and are always mentioned in the translation notes. The user can change or add to this algorithm simply by editing or resetting lambdacoms.

Index for Section A1

Appendix 2

The INTERLISP Interpreter

The flow chart presented below describes the operation of the INTERLISP interpreter, and corresponds to the m-expression definition of the LISP 1.5 interpreter to be found in the LISP 1.5 manual, [McC1]. Note that car of a form must *be* a function; it cannot evaluate to a function.

If car of a form is atomic, its function cell must contain

(a) an S-expression of the form (LAMBDA ...) or (NLAMBDA ...); or

(b) a pointer to compiled code; or

(c) a SUBR definition (see Section 8);

Otherwise the form is considered faulty.

If car of a form is an S-expression beginning with LAMBDA or NLAMBDA, the S-expression is the function. If car of the form begins with FUNARG, the funarg mechanism is invoked (see Section 11). Otherwise the form is faulty.

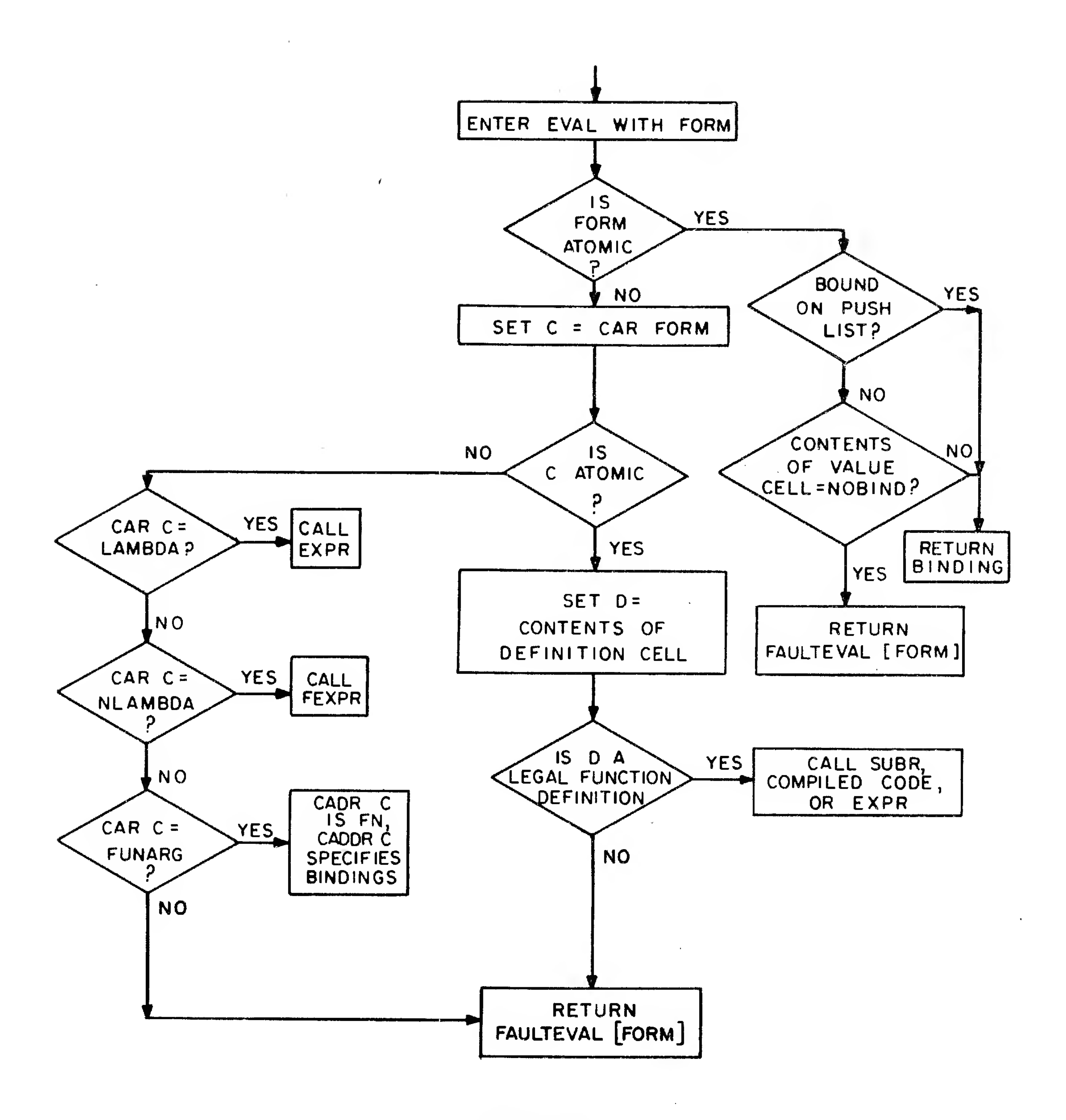


FIGURE A2-1

Note: variables c and d are for description only; they are not actually bound as variables.

Control Characters

Several teletype control characters are available to the user for communicating directly to INTERLISP, i.e., not through the read program. These characters are enabled by INTERLISP as interrupt characters, so that INTERLISP immediately 'sees' the characters, and takes the corresponding action as soon as possible. For example, control characters are available for aborting or interrupting a computation, changing the printlevel, returning to TENEX, etc. This section summarizes the action of these characters, and references the appropriate section of the manual where a more complete description may be obtained.

Control Characters Affecting the Flow of Computation

1. control-H — (interrupt) at next function call, INTERLISP goes into a break. Section 16.
2. control-B — (break) computation is stopped, stack backed up to the last function call, and a break occurs. Section 16.
3. control-E — (error) computation is stopped, stack backed up to the last errorset, and NIL returned as its value. Section 16.
4. control-D — (reset) computation is stopped, control returns to evalqt.
5. control-C — (TENEX) computation is stopped, control returns to TENEX. Program can always be continued without any ill effect with TENEX CONTINUE command.

If typed during a garbage collection the action of control-B, control-E, and control-D is postponed until the garbage collection is completed.

Typing control-E and control-D causes INTERLISP to clear and save the input buffers. Their contents can usually be recovered via the $BUFS (alt-modeBUFS) command, as described in Section 22.

I/O Control Characters

1. rubout — clears teletype input buffer. For example, rubout would be used if the user typed ahead while in a garbage collection and then changed his mind. Section 2. A bell is rung when the buffer has been cleared, so that the user will know when he may begin typing again.

Note: a sudden burst of noise on a telephone line frequently causes INTERLISP to receive a rubout, since the code for rubout is 177Q, i.e. all 1's. This causes INTERLISP to (mistakenly) clear the input buffer and ring a bell. If INTERLISP seems to be typing many spurious bells, it is a good indication that you have a bad connection.

2. control-O — clears teletype output buffer, Sections 2 and 14.

3. control-P — changes printlevel. Section 14.

4. control-A, Q — line editing characters, Sections 2 and 14.

5. control-R — causes INTERLISP to retype the input line, useful after several control-A's, e.g.,

```
user types:       ←DEFINEQ((LAMDA\A\DBA\Acontrol-R
INTERLISP types: DEFINEQ((LAMB
```

Miscellaneous

1. control-T — (time) prints total execution time for program, as well as its status, e.g.,

```
←RECLAIM()

GC: 8
RUNNING AT 15272 USED 0:00:04.4 IN 0:00:39
1933, 10109 FREE WORDS
10109
← IO WAIT AT 11623 USED 0:00:05.1 IN 0:00:49
```

2. control-S — (storage) change minfs. Section 10.

3. control-U — if typed in the middle of an expression that is being typed to evalqt, break1 or the editor, will cause the editor to be called on the expression when it is finished being read. See Section 22.

Index for Section A3

MASTER INDEX

Names of functions are in upper case, followed by their arguments enclosed in square brackets [], e.g. ASSOC[X;Y]. The FNTYP for SUBRs is printed in full; for other functions, NL indicates an NLAMBDA function, and * a nospread function, e.g. LISTFILES[FILES] NL* indicates that LISTFILES is an NLAMBDA nospread function. Words in upper case not followed by square brackets are other INTERLISP words (system parameters, property names, messages, etc.). Words and phrases in lower case are not formal INTERLISP words but are general topic references.

Page Numbers

Page Numbers

Page Numbers

Page Numbers

Page Numbers

Page Numbers

Page Numbers

Page Numbers

Page Numbers

Page Numbers

Page Numbers

Page Numbers

Page Numbers

Page Numbers

Page Numbers

Page Numbers

Page Numbers

Page
Numbers

Page Numbers

Page Numbers

Page Numbers

Page Numbers

Page Numbers

Page
Numbers